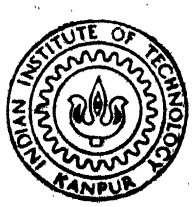
Design and Implementation of a 32 Bit VLSI RISC Architecture

by

Anil Kumar



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

MARCH, 1993

Design and Implementation of a 32 Bit VLSI RISC Architecture

A Thesis Submitted
in Partial Fulfilment of the Requirements
for the Degree of

MASTER OF TECHNOLOGY

By

ANIL KUMAR

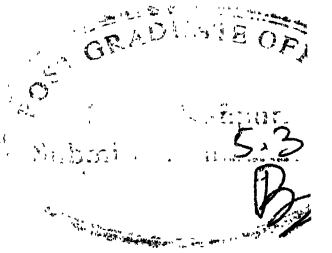
to the

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

March, 1993

CSE- 1993- M - KUM - DES

CERTIFICATE



This is to certify that the work contained in the thesis titled, **Design and Implementation of a 32 bit VLSI RISC Architecture**, was carried out under my supervision by **Anil Kumar** and it has not been submitted elsewhere for a degree.

March 1993
Kanpur

Rajat Moona
Assistant professor
Dept. of Comp. Sc. and Engg.
I.I.T., Kanpur

To
Amma & Pitaji

ACKNOWLEDGEMENTS

Designing a RISC processor is indeed a RISCy work. In the begining of this project, I was not very much sure about the completion in time but some how I was able to finish it.

I am grateful to my guide for his constant support and freedom he gave me during the most difficult part of this project. Some times we had heated discussion and I realized that "Boss was always right." Working under him is a memorable experience for me.

Many people assisted me during the prepartion of this thesis. In particular *Rgopa. Alok Kumar Gupta.* and *Raj kumar.* I also acknowledge all my classmates and other friends of mine from CSE and other departments for obvious reason. Last but not least, I appreciate my brother *Arun Kumar.* bhabhiji *Indu.* and their kids *Abhishek* and *Akanksha(Princie).* Without their motivation and inspiration this project might be still incomplete.

Anil Kumar

Abstract

RISC has become a mainstream movement to improve computing power and keep cost of design and design time low. In this thesis a 32 bit VLSI RISC architecture is designed and referred as iitk-RISC. The iitk-RISC supports four stage instruction pipeline and has 128 on-chip registers. This allows a fast operand fetch and simple data management by the compiler. The iitk-RISC also supports 4 Giga byte memory space organized in byte banks.

The processor has been implemented using 1.6μ scalable CMOS c3tu process. It has 83638 transistors and fits in the area of 7.85×7.24mm. The simulation results show that iitk-RISC can operate at 15MHz. With the large register set, it is possible to avoid memory reads and writes to approximately 10% of the program size. This gives an average performance of 14MIPS.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Many factors have influenced the design of a processor. These include availability of VLSI technology which made it possible to realize cache, data prefetch, data pipelines, instruction pipeline etc. on the processor chip. Using these concepts several processors (RISC, CISC and semi RISC) have been introduced in the recent past. Rest of this chapter discusses the concept of RISC, how VLSI made RISC a reality, "iitk-RISC", and the organization of the thesis.

1.1 Reduced Instruction Set Computer

In late 70s and early 80s general trend in computers was to increase the complexity of architectures by providing complex instructions and complex addressing modes. As a result the control unit of these processors occupied a good amount of silicon area: see for example, control unit of MC68000 is 68% of the chip area [5]. The complexity of these computers, (here onward referred as to CISC) had some negative consequences. These include increased design time and errors and increased duplication of resources, where resources stand for instruction set and functional units. Due to complexity of CISC even a million transistors chip was insufficient to design a single chip computer.

This led to a hypothesis that by reducing the number of instructions one can design a suitable VLSI architecture that uses a scarce resource, the chip area, more effectively than CISC [8]. Therefore the main emphasis in RISC design is to utilize the silicon area as optimaly as possible.

Some of the features of RISC design are as follows:

- They supports simple instructions. In RISC instructions are kept as simple as possible. In most of the cases a RISC instruction is equivalent to one micro call in CISC.
- RISC processors have fixed instruction format. A choice of fixed instruction format simplifies the instruction fetch and decode logic. Further registers access can be done in parallel with instruction decoding.
- There are good number of on-chip registers in RISC processor. This enables compiler to keep frequently used variables in registers.
- The core architecture of RISC is LOAD/STORE type architecture. Only LOAD and STORE instructions can access data from outside. Other instructions operate on registers as their source of operands and destination for result.
- All RISC processors execute one instruction per cycle.
- The instruction set in RISC processor has a good Support for high level languages.

Due to simple instructions, fixed instruction format, and small number of addressing modes the control logic of RISC is very simple. The silicon area thus saved is used to provide on-chip memory in the form of registers and caches. This reduces the demand on critically limited chip bandwidth. The result is increased system throughput. A typical RISC processor has up to 128 registers.

To utilize on-chip resources to their maximum potential the execution sequence is divided into several easy to implement subsequences. These execution subsequences are so arranged that a unit executing a subsequence is ready to accept a subsequence in every cycle. This arrangement of execution subsequences is called execution or instruction pipeline. For simplified and regularly running pipeline several things are to be considered. For example, in case of control transfer instruction, to keep the pipeline full instead of flushing, delayed branch technique is utilized. In almost all RISC processors the jump is not be taken until an intervening delay instruction has been executed. Advanced Micro Devices(AMD) claims that 90% of the time compiler can insert a useful instruction into the 'delay slot' [1]. In fact there is scope of other optimizations as well and this will improve the performance of the system many fold.

Instr	*i80386*	*MC68030*	*iitk-RISC*
cmp	4 cycles	7 cycles	1 cycles
call	17 cycles	17 cycles	1 cycles
ret	18 cycles	15 cycles	1 cycles
iret	22 cycles	20 cycles	1 cycles
jmp	17 cycles	14 cycles	1 cycles

Table 1.1: Instruction Execution Times for Some Processors

To summarize our discussion on RISC and CISC processors, the time required for some of the typical instructions is listed in table 1.1 for two CISC processors and compared with that required for these instructions in iitk-RISC.

1.2 Contribution of this Thesis

In this thesis, iitk-RISC has been designed and implemented at Computer Science and Engineering department of IIT KANPUR. It has almost all features of RISC discussed above. The iitk-RISC is a 32 bit architecture and consists of a control unit, an execution unit, a memory access unit, a write back unit, a register-file, and an interrupt handling unit. It supports 4 Giga byte memory space organized in byte banks. It has a four stage pipeline: Fetch/Decode and register fetch(IF and DRF), execution(EXEC), Memory access(MEM), and Write back(WB). The input clock frequency is 30MHz and it is divided by two internally to get the pipeline clock. In iitk-RISC an instruction cycle is taken to be same as one clock cycle. However, execution cycle is the number of cycles required to execute an instruction in a non-pipelined architecture. An execution cycle of iitk-RISC consists of four clock cycles.

The register-file of iitk-RISC consists of 128 registers of 32 bits each. The execution unit consists of a 32 bit adder/subtractor, a logic unit, and a barrel shifter. It is capable of doing 32 bit operations in a single cycle. The memory access unit can access a byte, a half word, and a full word from memory and data is aligned accordingly. In case of read access of memory the read data is converted to either signed 32 bit or unsigned 32 bit depending upon the type of read. The write back unit writes data into register-file or flag register or does not write as required by the instruction. The Interrupt handling unit monitors the on-chip illegal activities and interrupt requests from outside. For example illegal memory address. If an illegal activity or an interrupt request is detected then the interrupt handling unit

offest lines. The interrupt table of iitk-RISC is 2k bytes deep and lies at the bottom of the memory map.

The iitk-RISC instructions are of fixed size and fixed format. The instruction set of iitk-RISC has instruction for data movement among the register; instructions for arithmetic, logical, and shift operations; instructions to read an unsigned or signed byte, half word, and a word from memory; instructions to write a byte, a half word, and a word into the memory; and instructions to alter the control flow. Instruction set also includes few privilege instructions, for example, write into flag register and normal user is not permitted to execute these instructions.

1.3 VLSI CAD Tools

Several tools are available for designing and testing ICs by computer without actually fabricating them. These design tools consist of tools for simulating circuit at transistor and/or gate level, tools for designing layout and preparing mask for fabrication. Several fabrication technologies are available these days, for example, NMOS, SCMOS, HMOS, CMOS. A technology defines design rules and these rules are to be closely followed while designing the ICs. The design tools provide utilities to check the design for possible design errors. These rules include the minimum dimensions of layers, minimum separation of two layers etc. The design is normally done in terms of scalable unit λ as defined by C. A. Mead and L. Conway [6].

The iitk-RISC is designed with the aid of *nelsis-IC design tool* [7], from TU Delft software distribution available in Computer Science and Engineering Department, IIT KANPUR. The layout of iitk-RISC is designed using interactive layout editor **dali** with λ fixed at 0.200μ, for 1.6μ technology. All circuit simulations are done with **SLS** (A Switch Level Simulator) [4] and **SPICE** [10].

1.4 Thesis Organization

Chapter 2 reviews some of the existing true RISC and semi RISC architectures. The next three chapters deal with the instruction set, architecture, and layout of iitk-RISC. They show how iitk-RISC fits into the category of true RISC.

The thesis concludes with the results that have been obtained and some possible extensions to the iitk-RISC. In Appendix A instructions set of iitk-RISC is given. The basic modules, that are used in implementation of iitk-RISC, are illustrated in Appendix B.

Chapter 2

A Survey of RISC Mainstream

This chapter discusses some of the RISC processors' design. The processors discussed include: RISC I, RISC II, MIPS R2000, intel's 80960, AMD's 29000, and SPARC. The processors are discussed roughly in ascending order of their architectural complexity. The discussion is brief and the design issues to be considered in subsequent chapters are included specifically.

Some of the newly introduced processors have been excluded because these processors do not qualify the basic criterion of RISC, namely, simplicity. Though these are claimed to be RISC processors by their respective vendors, yet computer architects prefer to call these processors *Pseudo RISC*.

2.1 The Berkeley RISC I

The concept of RISC is not new but it was taken seriously only after the RISC I was designed at UC Berkeley. The RISC I architecture [8, 5] has 31 instructions, most of which do simple ALU and shift operations on registers. Instructions, data, addresses, and registers are 32 bits wide. RISC I execution cycle consists of fetching instructions and execution of instructions. With two stage pipeline no internal forwarding is required. The register-file consists of 138 registers of 32 bits each, though its first model "RISC GOLD" was fabricated with 72 registers only. Separate buses are used for reading two registers and writing result back. Register-file is partioned into overlapped register-windows of 32 registers. Out of these 32 registers, $R_0 \ldots R_9$ are termed as global registers, $R_{10} \ldots R_{15}$ are termed as low registers, $R_{16} \ldots R_{25}$ are termed as local registers, and $R_{26} \ldots R_{31}$ are termed as high registers. Every time a CALL instruction is executed a new window is allocated and the low registers of caller's window become the high registers of callee's window. On overflow or underflow a

trap is generated and necessary steps are taken by the interrupt handler routine to allocate registers in memory. For regular and smooth running of pipeline the concept of delayed branch is used in case of control transfer instructions.

The layout of RISC I was designed using NMOS technology with λ at 2 microns and no buried contacts.

2.2 The Berkeley RISC II

This is the successor of RISC I and most of its architectural features were copied from RISC I. The aspects in which it differs from RISC I are discussed in this section.

The RISC II CPU [5] has two modes of operation namely user-visible(u-v), and interrupt-handling(i-h). Some of the instructions can be executed in i-h mode only. In case of illegal execution of instructions a trap is generated.

RISC II is a 32 bit architecture. The instruction execution pipeline consists of three stages: fetch, compute(ALU), and write back result. The simultaneous read and write to the register-file is avoided and only two bus register-file is used. The register-file consists of 8 overlapped windows of 32 registers each of which $R_0 \ldots R_9$ are global registers, $R_{10} \ldots R_{15}$ are input registers, $R_{16} \ldots R_{25}$ are local registers, and remaining $R_{26} \ldots R_{31}$ are output registers. On call instruction a new window is allocated and the current status of window is reflected by CWP(current window pointer). and SWP(saved window pointer). These two window pointers are the part of PSW(program status word). PSW includes four condition flags (zero, negative, overflow, and carry), interrupt enable bit(I), system mode bit(S), and previous system mode bit in addition to the CWP and SWP. Three program counters are used, NXTPC contains the address of next instruction to be executed. PC contains the address of the instruction that is in execution process, and LSTPC contains the address of interrupted instruction in case of interrupt. Whenever an illegal condition is detected an interrupt is raised and content of PC is pushed into LSTPC.

Three types of instruction formats are supported. These are *long immediate*, *short immediate*, and *register-register*. The instruction set consists of 53 instructions. It includes instructions for arithmetic and logical operations; instructions for loading and storing a byte, a half word, and a word, control transfer instructions, and some other miscellaneous instructions.

The layout of RISC II was designed using NMOS technology with λ at 2 microns and it requires a 4 phase clock.

2.3 The MIPS R2000

The MIPS R2000 [9] is a 32 bit architecture. The instruction execution pipeline consists of five stages: IF(instruction fetch), RF(register fetch), MEM(data memory reference), WB(write result back). The CPU is logically composed of six synchronized units: Master pipeline control unit, Execution unit, Address unit, Translation look-aside buffer, System co-processor interface unit, and External interface controller. The Master pipeline control latches instruction field off the data bus and performs instruction decode. It also controls the pipeline if any abnormal conditions arise.

The execution unit is composed of a 32 bit shifter, an incoming data aligner, an arithmetic logical unit and, a multiply and divide unit. Multiply and divide unit operates autonomously from rest of the processor. Internal bypassing is done, to get the consistent result, inside the execution unit.

The R2000 does not use condition codes to govern the control flow. Instead, the CPU provides branches based directly on simple data comparison and ALU operations that directly create boolean values in register. The compiler synthesizes branch conditions from the instructions available to access these boolean values. Delayed branch concept is used to avoid bubbles in pipeline.

All the memory reference instructions are processed in MEM part of the execution cycle. The address of data is computed in ALU part of execution cycle and data is actually accessed in MEM part of execution cycle. For instructions other than memory access instructions MEM part of execution cycle is converted into an intermediate latency. The layout of R2000 was designed using double metal CMOS technology with λ at 2 microns.

2.4 The 80960 RISC Architecture

Architecturally, 80960 architecture [2] is very different from RISC mainstream. For one thing, its design has microcode in a number of places with the main 32×42 bit microcode ROM which is used to generate 42 control signals. For another, it has a full IEEE-754 floating point unit complete with its own 80 bit registers. Its instruction set includes a

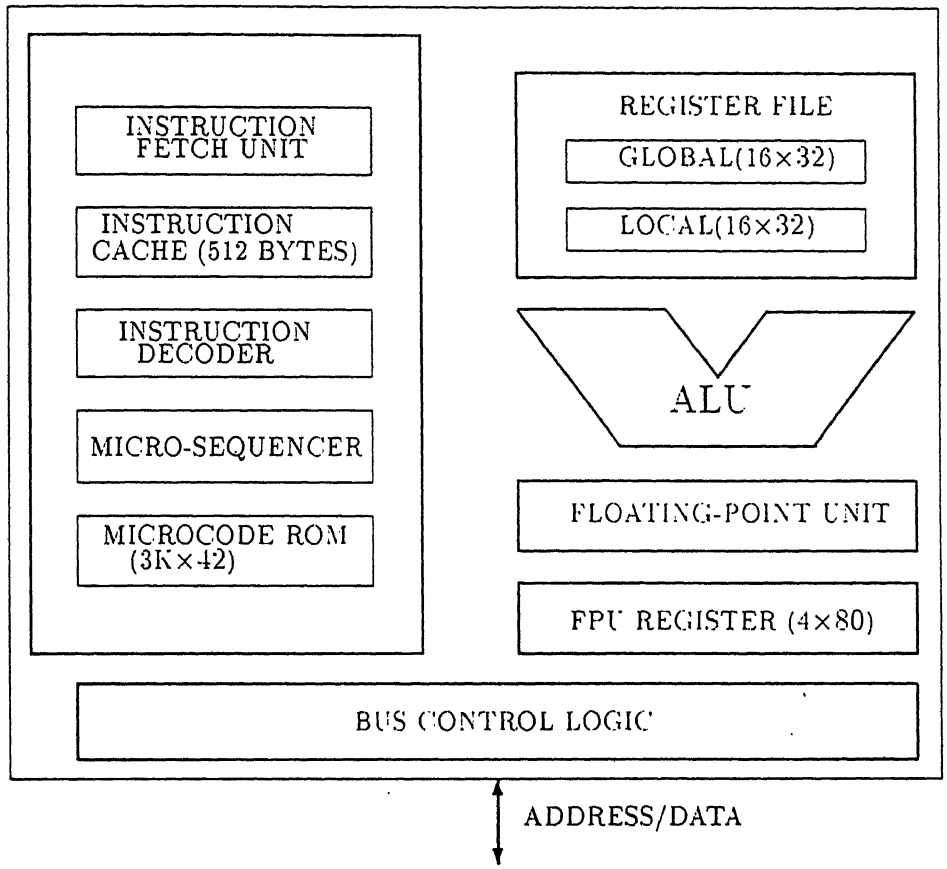


Figure 2.1: Intel's 80960 RISC Architecture.

large number of specialized controller-type instructions like boolean logic operations and bit manipulation. Some of the architectural details are shown in figure 2.1. It was fabricated with 1.5μm using CMOS technology.

2.5 The 29000: AMD's RISC Machine

The computer architects consider CISC as a computer running inside computer. Dan O'Dowd, a computer architect who was responsible for 32000 CISC family from National Semiconductor Corp., points out that a RISC performance can be extracted from a CISC computer by removing the outer "macrocomputer" level from a CISC, leaving the inner microcoding "computer within a computer." On this idea AMD introduced this bit slice RISC machine. It was fabricated using 1.2μ CMOS technology.

The architecture of the 29000 is shown in figure 2.2. It is a 32 bit architecture and has 192 on-chip registers [2].

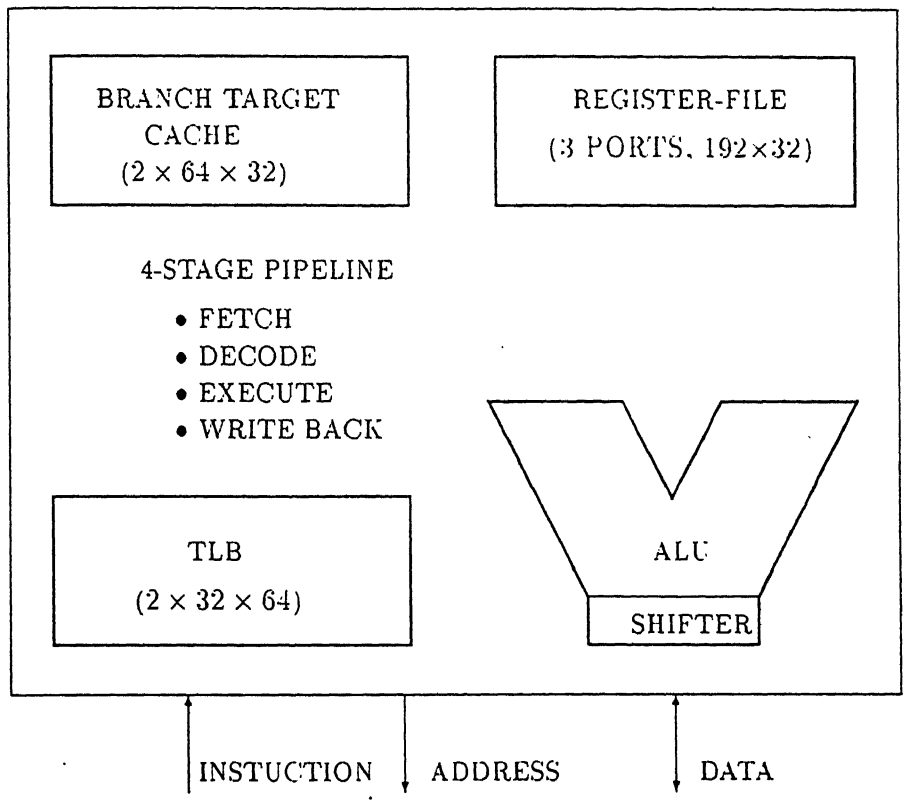


Figure 2.2: The AMD's 29000, a bit slice RISC Architecture.

2.6 The SUN's SPARC

SPARC (*Scalable Processor Architecture*) [2] was intended to be a high speed true RISC processor unlike its contemporary architectures. The basic architecture of SPARC is shown in figure 2.3. The first SPARC was realized in a 20000-gate, 1.5μm gate array from Fijitsu Microelectronics Inc. It is a 32 bit machine and has 136 on-chip registers. These registers are partioned into overlapped windows and a window is assigned to a task. It was later fabricated using 0.8μm CMOS technology.

2.7 Conclusion

The concept of RISC processors started with the RISC I architecture and its on-chip register resources, small number of addressing modes, and small number of instructions were publicized as a RISC standard. However, the new RISC processors are departing from this standard very much. The new RISC processors seem to follow a different line of design, closer to their counterpart CISC. In-fact people prefers to call these RISC processors *Pseudo RISC*. Some of the interesting observations about these processors are as follows:

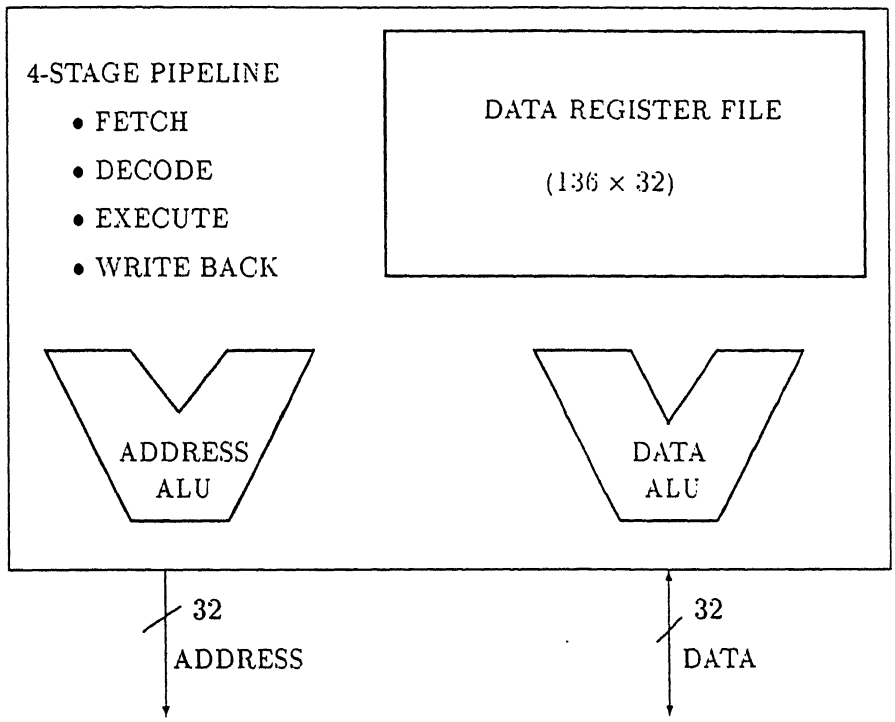


Figure 2.3: Sun Microsystems' SPARC Architecture.

- All are 32 bit architectures.
- All are pipelined architectures and use a 2 to 4 stage pipeline.
- The number of on-chip registers is declining as some extra features like MMU, FPU etc. are added on-chip.
- All architectures use on-chip or off-chip data and instruction caches.
- The complexity of processors is increasing.
- The chip area required for fabricating the processors is increasing with time.

Chapter 3

Instruction Set of iitk-RISC

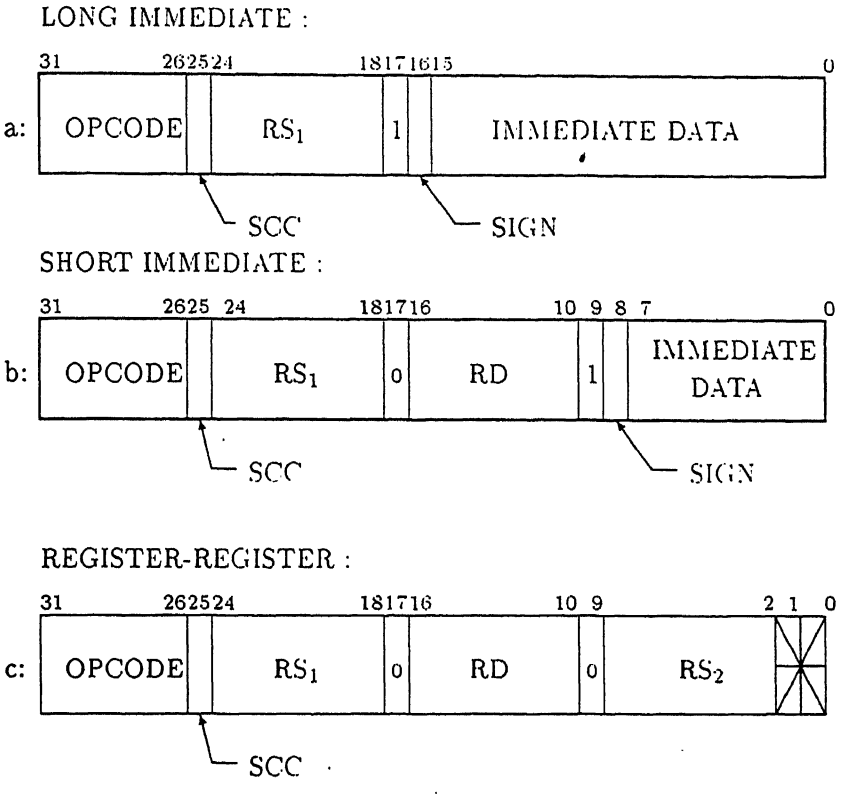
There is always a trade off between compactness of code and CPU performance. As memory is a critical resource, a compact code will enable smaller devices for handling the same amount of compiled code. Here memory devices include secondary memory, primary memory, and instruction cache.

There are two main methods for reducing the average code size. Firstly, an instruction format closer to Huffman encoding may be utilized. This means having a variable number of fields in the instructions. The choices are made according to the relative usage of instruction and fields. Secondly, the approach followed in CISC is that of merging two or more instruction into one [5].

Both the approaches have their own merits and demerits. We shall not discuss that here. The compact code also has some negative consequences, discussed in section 1.1.

In accordance with the RISC philosophy, all iitk-RISC instructions are one word long and have fixed format for easy decoding. The meaning of some fields of in an instruction format is fixed and can not be altered. The iitk-RISC is a 32 bit architecture and can support 4 Giga-byte of memory space. It supports a LOAD/STORE type architecture where a memory access is done by some specified instructions. Data in memory can be stored as a byte, a half word, and a full word. The accessed data is converted into signed or unsigned full word before writing into register-file. This offers simplicity and full flexibility of supporting different data types.

This chapter discusses the instruction set of iitk-RISC briefly. A detailed description can be found in Appendix A. The instruction set is divided into four broad categories: *Data Movement Instructions*, *Data Manipulation Instructions*, *Control Transfer Instructions*, and *Miscellaneous Instructions*.



SCC: SET CONDITION CONTROL

RS_1, RS_2: REGISTER OPERANDS

RD: DESTINATION REGISTER

SIGN: SIGN BIT FOR IMMEDIATE DATA

Figure 3.1: Instruction Formats of iitk-RISC.

3.1 Instruction Format

The different instruction formats of iitk-RISC instructions are shown in figure 3.1. All instructions of iitk-RISC are one word long and aligned at word boundaries in memory. The format of an instruction in the instruction set depends on the addressing modes it supports.

3.2 Addressing Modes

The iitk-RISC in accordance with the philosophy of RISC architecture supports small number of addressing modes. Some fields in the instruction format carry same information for all instructions whereas some other fields encode the information specifying the addressing mode used and hence are dependent on addressing modes. For control transfer instruction

relative addressing mode is provided and it has been simplified enough for the implementation purpose. All instructions use at least one operand which is specified in the field RS_1.

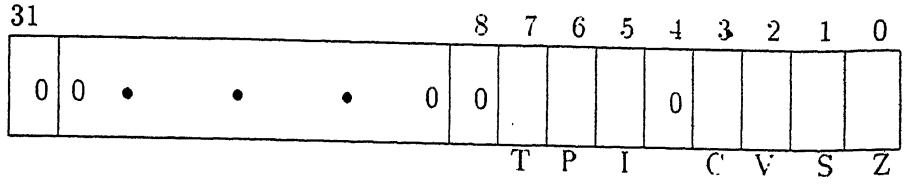
Register-Addressing The second source of two operands instruction is a register. This addressing mode is supported by all two operand instructions. Format for these instruction is shown in figure 3.1.c.

Long-Immediate Addressing Second source of the instruction is 17 bit signed immediate data. The sign of immediate field is determined by SIGN(bit<16> of Long-Immediate Instruction format) and immediate data is sign extended to 32 bit. This addressing mode is supported by two operand instructions with implicit destination. Format of these instructions is shown in figure 3.1.a.

Short-Immediate Addressing Second source of the instruction is 9 bit immediate data. The sign of immediate field is determined by SIGN(bit<8> of Short-Immediate Instruction format) and immediate data is sign extended to 32 bit. This addressing is supported by all two operands instructions. Format of these instructions is shown in figure 3.1.b.

Register-Relative Addressing The effective address(EA) is calculated as $EA=[RS_1]+S_2$, S_2 can be a register, 16 bit immediate offset, or 8 bit immediate offset. If S_2 is immediate offset then it is first converted to 32 bit signed immediate offset. This addressing mode is supported by all the control transfer and LOAD/STORE instructions.

PC-Relative Addressing If PC is used to calculate the effective address in place of RS_1, then Register-Relative addressing becomes PC-Relative. The effective address is calculated as $EA=[PC]+S_2$. S_2 can be a register, 17 bit immediate offset, or 9 bit immediate offset data. Similar to previous addressing modes, S_2 is first sign extended to 32 bit. In this addressing mode RS_1 is ignored because in effective address calculation PC is used as first operand of the instruction and RS_1 field of instruction can not be used as other then the first operand of the instruction. This addressing is supported by PC relative control transfer instructions.



Z: ZERO FLAG S: SIGN FLAG
V: OVERFLOW FLAG C: CARRY FLAG
I: INTERRUPT FLAG P: PRIVILEGE FLAG
T: TEMP FLAG

Figure 3.2: The Flag Register of iitk-RISC.

Data Manipulation Instructions do not support Long-Immediate Addressing as in this case the destination can not be specified.

3.3 Flags

The flags in iitk-RISC are arranged in a form of a 32 bit register and can be accessed by the user. It is the implicit source/destination for two instructions. The flag register of iitk-RISC is shown in figure 3.2.

The C, V, S, and Z flags are condition flags and reflect the result of data Manipulation instructions and flags are changed according to explain below if SCC field in the instruction is set to 1 (bit<25>=1) and instruction can change the flags.

Z:=[DEST==0]; S:=DEST<31>

V:=0; in case of shift and logical instructions.

V:=[32-bit 2's-complement overflow occurred]; in case of arithmetic instructions.

C:=0, in case of logical instructions.

C:=RS_1 <shift_count-1>((shift_count-1)>0); in case of shift right instructions.

C:=RS_1 <32-shift_count>((32-shift_count)>0); in case of shift left instructions.

C:=carry<31>to<32>(for RS_1 and S_2 unsigned): in case of addition instructions.

C:=NOT[borrow<31>to<32>](for RS_1 and S_2 unsigned); in case of subtraction instructions.

The I, P, and T flags are used for system control. If I=0, no interrupt will be honored by the CPU. If P=0 then CPU is in privilege mode. The T flag is used to store the content of P in case of an interrupt. A normal user can access the flag register however. changing the flag register is allowed only in privilege mode. When the system is reset all flags are cleared

except T flag which is set. The system flag T is set because at the end of the execution of reset program a *reti* instruction will be executed to transfer the control to some user program.

3.4 Instruction Set

In this section, the instructions have been grouped into five categories and the discussion of a particular category is applicable to all instructions of that category.

3.4.1 Data Movement Instruction

The only instruction that comes in this category is *mov*. In this instruction second source is redundant and ignored. No flag is changed even if SCC=1 because in moving content of one register to some other register no arithmetic, logic, or shit operation is involved.

3.4.2 Data Manipulation Instructions

This category includes instructions for addition, subtraction, logical operations and shift operations. All of these support Register-Addressing, and Short-Immediate Addressing. All the instructions except *inv* are two operand instructions. Instruction *inv* takes only one operand. Flags are changed to reflect the result of Data manipulation instruction if SCC bit is set.

For shift instructions shift count is provided by the second operand. As the shifts more then 31 bit are meaningless only. 5 bit(<4:0>) are used as shift count. Various types of shift supported in iitk-RISC are illustrated in figure 3.3.

3.4.3 LOAD/STORE Instructions

This set of instructions is used to access memory for read and write operations. Data can be accessed as a byte, a half word, and a full word. The width of the data being accessed is provided by two data-width-lines: W_0 and W_1.

In case of *load* instructions CPU always internally reads full word. Separate load instructions are provided for loading unsigned byte, half word, and full word. Similarly for loading signed byte, half word, and word separate instructions are provided. If read data is byte or half word then it is aligned and converted to either signed 32 bit or unsigned 32

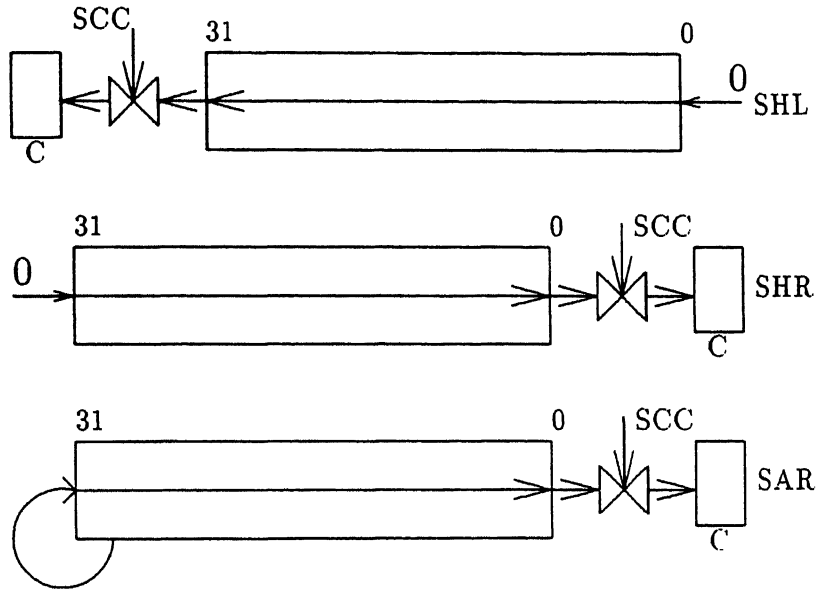


Figure 3.3: Shift Operations.

bit, depending on the signed or unsigned access of data, before writing into register. However, in case of full word read operation data is written into register specified without any modification. An example of loading second signed byte is shown in figure 3.4

The *load* type instructions support Register-Relative Addressing mode. However, the second field S_2 in effective address calculation can not be 16 bit immediate offset because load type instructions require a destination register.

For write access of memory, separate instructions are provided for storing a byte, a half word, and a full word. The iitk-RISC CPU always outputs a full 32 bit word onto the bus. However, only some of the bytes in that word are to be written into the corresponding bytes of addressed word. The number of bytes to be written are determined by the instruction in execution. For example, in case of *storb* instruction only one byte will be written. For this iitk-RISC supports a memory system organized in byte banks. The bank(s) in which byte(s) is(are) to be written is(are) selected by W_0 and W_1, A_0 and A_1 lines of address bus. The width code lines indicate the width of item to be written. The decoded meaning of these four lines is given in table 3.4.3. Figure 3.5 shows an example of storing a half-word into effective-address <1:0>=10

The store type instructions support Register-Relative Addressing mode where the effective address is calculated as:eff-address=[RS_1]+0.

No instruction from this category modifies the flags irrespective of SCC bit.

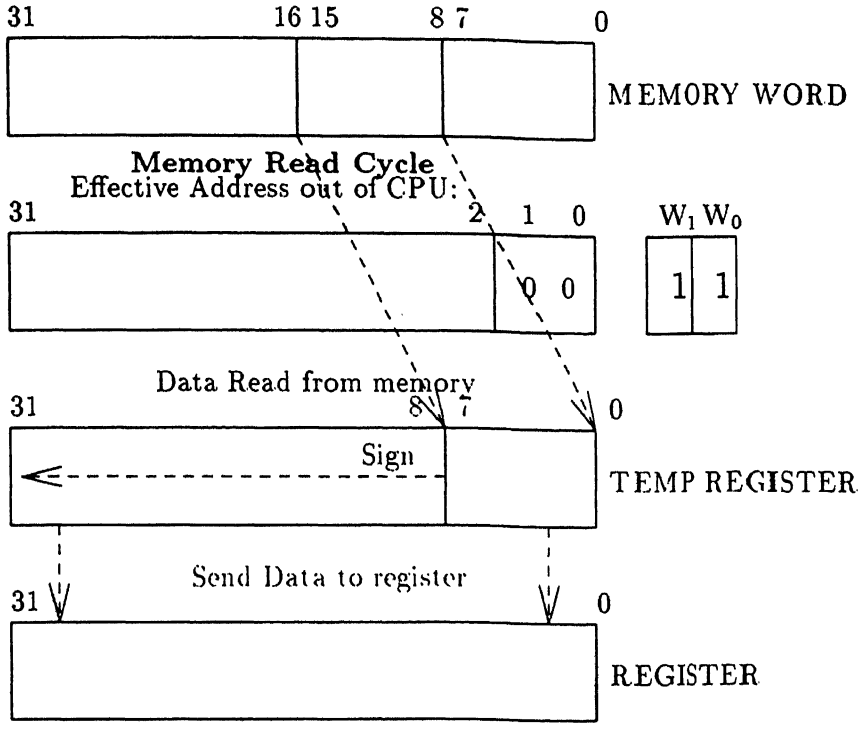


Figure 3.4: Loading a Byte.

INSTR	OUTPUT INFORMATION FROM CPU				MEMORY BYTE BANKS THAT SHOULD BE ENABLED			
	WIDTH CODE		eff-addr<1:0>					
	W_1	W_0	A_1	A_0	3 <31:24>	2 <23:16>	1 <15:8>	0 <7:0>
storb	ON	OFF	0	0				E
			0	1			E	
			1	0		E		
			1	1	E			
storh	OFF	ON	0	0			E	E
			1	0	E	E		
storw	ON	ON	0	0	E	E	E	E
load type	ON	ON	-	-	E	E	E	E

Table 3.1: Storing Data of Various Widths.

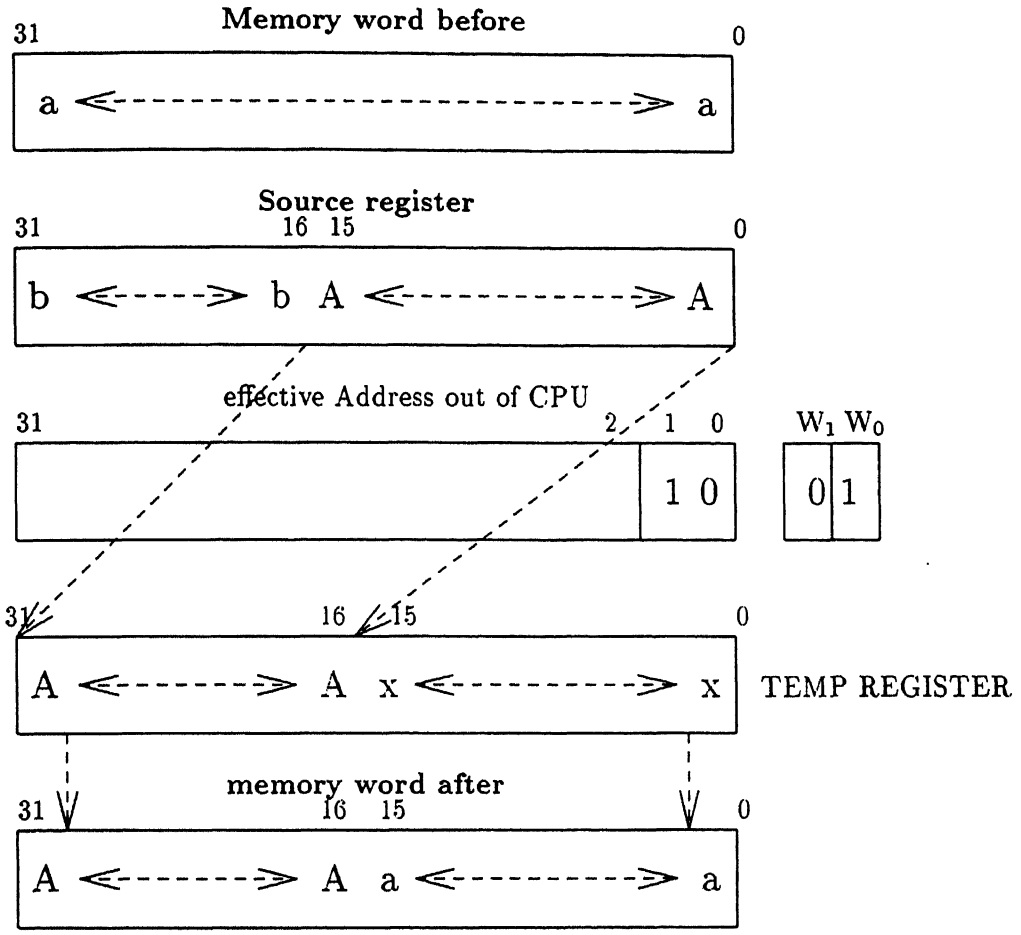


Figure 3.5: Storing a Half-Word.

3.4.4 Control Transfer Instructions

Conditional jumps, Unconditional jumps, and Procedure call constitute this group. This set of instructions is used to translate decision boxes of flow chart of a program. There are two types of control transfer instructions provided. The first type of instructions support Register-Relative Addressing mode whereas second type of instructions support PC-Relative Addressing. The jump conditions are evaluated according to the table 3.2. A jump delayed by one cycle takes place if the condition evaluates to TRUE. The instruction slot next to the control transfer instruction is called *delay slot* of that control transfer instruction and the instruction in delay slot is always executed due to delayed control transfer.

The *call* instruction differs from other instructions of this group in one respect. It pushes the address of the instruction that has been executed in delay slot into specified register. It will be used as a return address. Thus to get the correct return address the address pushed by *call* instruction should be incremented by 04h. This can be done in the control transfer

INSTR	CONDITION TEStED	INSTR	CONDITION TESTED
jgtr,jgtp	$(S \oplus V) \vee Z$	jler,jlep	$(S \oplus V) \vee Z$
jger,jgep	$\overline{S \oplus V}$	jltr,jltp	$S \oplus V$
jhir,jhip	$\overline{C \vee Z}$	jlosr,jlosp	$\overline{C \vee Z}$
jncr,jncp	$\overline{C}$	jcr,jcp	C
jpr,jpp	$\overline{S}$	jnr,jnp	S
jner,jnep	$\overline{Z}$	jer,jep	Z
jnvr,jnvp	$\overline{V}$	jvp,jvr	V
jmpr.jmpp	1		

Table 3.2: The iitk-RISC Jump Conditions.

instruction used to return from procedure or function call. Figure 3.6 shows an example.

Instructions of this group do not modify the flags even if SCC bit is set.

3.4.5 Miscellaneous Instructions

In this group two types of instructions are included. Instructions *reti*, *getlpc*, and *putpsw* are privileged instructions and can be executed only if P flag is 0. The non-privileged group comprises *getpsw* and *nop* instructions.

The *reti* instruction is used for return from interrupt handling routine. It restores the previous system operation mode(P flag) and loads the PC with the content of specified register. The control transfer is delayed by one cycle.

The *getlpc* must be the first instruction of any interrupt handling routine. It moves the content of LSTPC (the address of interrupted instruction) to register specified, which will be used to restart the interrupted instruction.

The *putpsw* instruction is used to change the content of flags. Changing the flags C, S, Z, and V is meaningless because these flags change dynamically and no one can predict the content without actually analyzing the program. The result of instruction will be effected only after the end of execution cycle.

The *getpsw* pushes the content of flags into the register specified. The *nop* instruction does nothing and generally used to fill the delay slot if compiler is unable to fill it with some meaningful instruction. Some times it used to introduce calculated amount of delay in program.

No instruction of this group except *putpsw* changes the flags. *putpsw* changes flags even if SCC bit is clear.

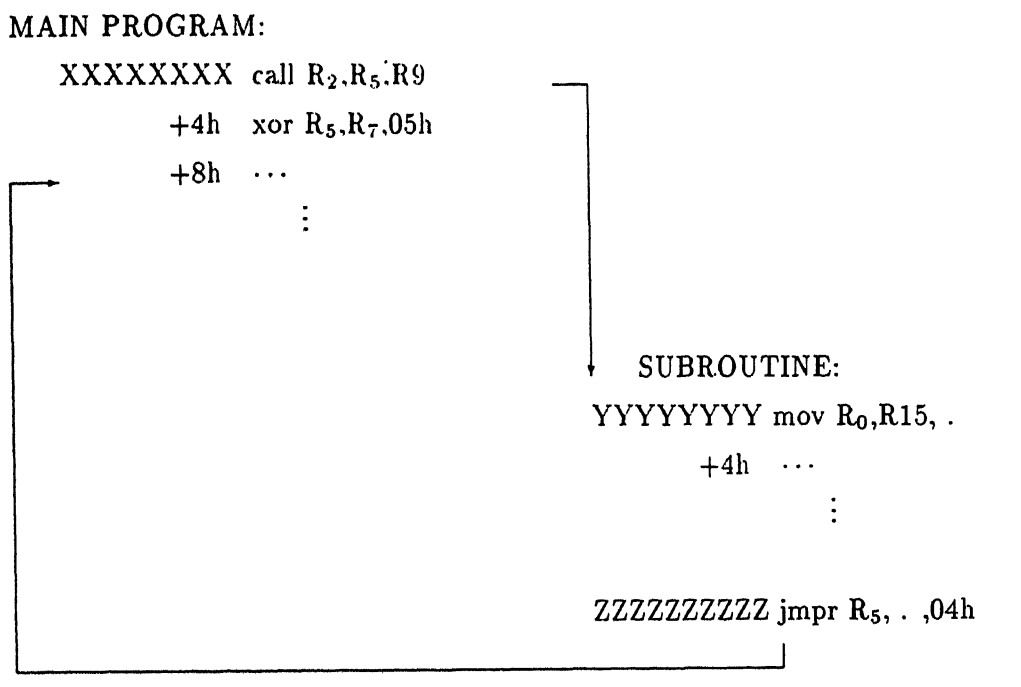


Figure 3.6: An Example of Procedure Call.

Illegal Instruction : FFFFF808h
Address Violation : FFFFF800h

Figure 3.7: Interrupt Vector for Illegal Conditions.

3.5 Illegal Conditions

There are some illegal conditions that may arise during the program execution. We have tried to detect all such illegal conditions that may affect the program execution. On detection of an illegal condition the system changes its operating mode to privilege mode and an interrupt or exception handling routine is called. The instruction which causes the illegal condition is executed as *nop*. Figure 3.5 shows the interrupt vector for the two case of illegal conditions to be discussed.

These illegal conditions are: Illegal Instruction and Memory Address violation(or Illegal Memory Address).

3.5.1 Illegal Instructions

There are three cases of illegal instruction. These are illegal opcode, privilege violation, and invalid field. An illegal instruction is executed as *nop* and an interrupt is raised.

Illegal Opcode

The opcode field of iitk-RISC's instruction is 6 bit wide. With 6 bit 64 combinations are possible. For iitk-RISC some of these combinations are meaningless, that is, no instruction is associated with them. If iitk-RISC CPU finds any of these combinations, it is treated as Illegal Instruction. The list of illegal combinations is as follows: 17h, 1Ch, 1Dh, 1Eh, 1Fh, 20h, and 30h.

Privilege violation

Some of the instructions are protected from the execution by a normal user because execution of these instruction may cause some serious problems during execution. To avoid execution of these instruction by a normal user privilege flag(P) is checked before executing these instructions. If the check succeeds then only instruction is inserted into pipeline for execution otherwise instruction is executed as *nop* and instruction treated as an illegal instruction.

Invalid Fields

Sometimes the CPU finds that a particular field of a instruction is missing, for example, destination register is not given. This is another case of illegal instruction and treated accordingly.

3.5.2 Memory Address Violation

To access a full word and a half word the last two bits of effective-address(<1:0>) must be 00 and 00/10 respectively. Because, in case of iitk-RISC words are aligned at word boundaries and half word at half word boundaries. If any memory request violates this restriction, the address will be corrected by converting these two bits to 00. Memory will be accessed using this address and an interrupt is raised.

3.6 Evaluation of the iitk-RISC Instruction Set

In This section we evaluate the iitk-RISC instruction set. We discuss its appropriateness for High Level Languages(HLL) and its impact on code size.

HLL Statement:	iitk-RISC instructions:
if (count++≤n)	sub R_n, R_0, R_{count};SCC set jge *target_addr* $R_{count} \leftarrow R_{count} + 1$
C->xyz==*abc	load $R_{t1} \leftarrow M[R_C + OFFS_{xyz}]$ load $R_{t2} \leftarrow M[R_{abc} + 0]$ sub R_{t1}, R_0, R_{t2};SCC set
C=C->xyz	load $R_C \leftarrow M[R_C + OFFS_{xyz}]$

Figure 3.8: HLL Translation into iitk-RISC Codes.

3.6.1 Instruction Set and High Level Language

In most of the cases iitk-RISC instructions are similar to a micro-instruction of a typical CISC computer. One can argue that the instruction set of iitk-RISC is "of too low level" for a High Level Language.

However, several frequently used statements of HLL can be compiled into only a single or a few iitk-RISC instructions. Some examples are shown in figure 3.8

Thus, iitk-RISC instructions are not far away from some very frequently used HLL statements. Also the variants of LOAD/STORE as discussed earlier enable the iitk-RISC to support almost all the data types without much overhead.

3.7 Conclusion

The instruction set of iitk-RISC is designed in such a way that every instruction can be executed in a single cycle but not at the cost of increasing the number of instruction required to translate a HLL statement. The code translated in simple instructions has more scope of optimization as compared to translated in complex instructions. Further, we feel 4 Giga byte memory space is enough to fit any program of these days. Also iitk-RISC has enough number of on-chip registers for compiler to keep almost all variables. Based on these considerations, in case of iitk-RISC relatively uncompacted code will not cost too much. In a nutshell we can say that the performance of iitk-RISC will be better then most of the CISC computers.

Chapter 4

The iitk-RISC Architecture and Pipeline

This chapter discusses the architecture of iitk-RISC including instruction pipeline of iitk-RISC, the delayed control transfer, and the internal forwarding mechanism. The basic timing for read, write, and fetch cycles of iitk-RISC is also discussed. The discussion of architecture and micro-architecture is intermixed for easy understanding.

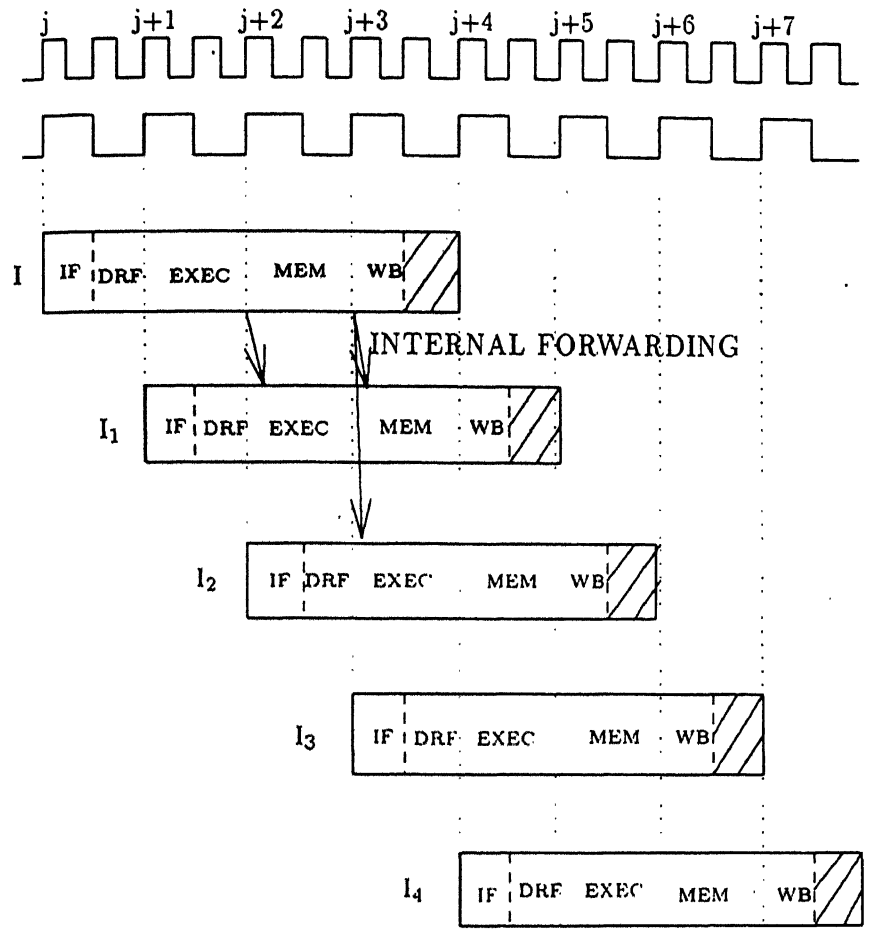
4.1 The Instruction Pipeline of iitk-RISC

The instruction pipeline of iitk-RISC is a four stage pipeline and supports internal forwarding of data for inter-stage data dependency resolution. In this section we focus on the instruction pipeline and its impact on the iitk-RISC architecture.

4.1.1 A Four Stage Pipeline

Execution sequence of a simple instruction in a processor follows a very simple pattern. It consists of the following sequence of operations: Fetch Instruction, Decode instruction, Fetch operands, Perform operation, and finally write result into destination. These operations do not require equal amount of time to finish. As a pipeline is more effective if all of its stages take roughly equal amount of time, some operations can be combined to provide operations of roughly equal time. For example instruction decode and fetch operands are combined in iitk-RISC.

We assume that iitk-RISC will have an off-chip instruction cache and fast primary memory for filling the instruction cache and data memory access. This enables fast instruction fetch. Further, some of the operations can not be made faster because of implementation



IF: INSTRUCTION FETCH DRF : DECODE and REGISTER FETCH

EXEC : EXECUTION MEM: MEMORY REFERENCE

WB: WRITE BACK RESULT

Figure 4.1: Instruction Pipeline of iitk-RISC.

limitations and high cost of implementation. For example, the implementation cost of carry lookahead adder increases exponentially if we go for a faster adder. Due to these reasons instruction decoding could not be clubbed with execution phase. In iitk-RISC, to make all the steps of the execution sequence of almost equal time, fetch instruction and decoding is done in the same cycle. Operands are fetched in parallel with instruction decoding. The basic timing for fetch cycle, to be discussed later, validates our assumption.

Considering all above assumptions we have designed the iitk-RISC pipeline. It is a 4 stage pipeline and shown in figure 4.1

The instruction is fetched during IF part of jth cycle. After fetching the instruction

following actions are taken in parallel:

1. Opcode part of the instruction is decoded.

2. Immediate data/offset is aligned and signed extended to 32 bit immediate data/offset. The 32 bit data/offset is stored in immediate register.

3. Register operands are fetched.

For fetching register operands and taking immediate data/offset from the instruction, it is assumed that an instruction has second register operand and 17 bit or 9 bit immediate data/offset field. The first register operand is present in all the instructions. The bit<8:2> of an instruction are used for register fetch even if bit<9> is 0. The bit<9> and bit<17> determine the presence or absence of second register operand. The bit<16:0> are taken as long immediate data if bit<17> is set. If bit<17> is 0 then the bit<8:0> are taken as short immediate data. The operands to be used for an instruction are determined by the control signals generated by decode of an instruction as discussed in Section 3.4.1. After the decoding phase is over the necessary operand sources are enabled. The RS_1 is required by all the instructions as first operand whereas either RS_2 or immediate data/offset is required as second operand.

In (j+1)th cycle, the decoded instruction together with all the necessary operands is sent for execution. In execution phase of the execution cycle the execution unit performs the required operation on the operands of the instruction. The operation performs by the execution unit may be add/subtract, logical operation, or shift operation. At the end of (j+1)th cycle all flags are updated if SCC bit is set in the instruction and the instruction can modify the flags.

After the execution stage gets over, memory access stage reads the result of the execution stage. If instruction under consideration involves memory access then a memory access cycle is initiated in (j+2)th cycle, otherwise memory access stage sits idle in (j+2)th cycle. At the end of (j+2)th cycle it sends data to the write back unit, which may be data read from memory, result of the execution unit, or nothing depending on the instruction. For example in case of load type instruction memory access unit passes the accessed data, in case of data manipulation instruction it passes the data read from the execution unit, and in case of control transfer instructions nothing is passed. Inclusion of this stage as a regular feature

of the pipeline requires some explanation. In RISC II and some other processors, in which memory stage is not a regular feature of pipeline, memory access stage is included only when there is a memory access instruction in pipeline. We found that in such processors decision regarding the inclusion of memory stage is to be made either during decode of instruction or at the end of execution cycle and it should be known to write back stage well in advance so that write back unit can delay its action by one cycle. This increases the number of control signals and affects the regularity of the pipeline.

In write back stage of pipeline, data read from memory access unit is written into appropriate destination if required by the instruction. The destination may be a register in register-file or flag register.

4.1.2 Analysis of Pipeline

A pipeline also has some negative effects on the performance of the processor if the architecture is not designed carefully. In this subsection we have discussed the possible problems with a 4 stage pipeline of iitk-RISC and how these problems are taken care of.

Pipeline Suspension During Data Memory Access

If there is a memory access instruction already in pipeline, then a instruction can not be fetched during the cycle when this load instruction will be processed by memory access unit in the computer systems in which address and data buses are shared by instruction fetch and data access both. This is because the address and data buses are busy in doing data memory access operation in that cycle. This situation is shown in figure 4.2

iitk-RISC in this situation pushes one bubble into the pipeline, by inserting a *nop*. There are several remedies to avoid this bubble inside the pipeline. Firstly, separate buses for data memory access and instruction fetch may be used. This choice though promising, requires a two port external memory and increased pin-count. However, the instruction fetch will never clash with data memory access. Secondly, data cache may be used. In this case time shared address and data buses can be used, that is, address and data buses will be used to fetch the instruction and data memory access in the same cycle.

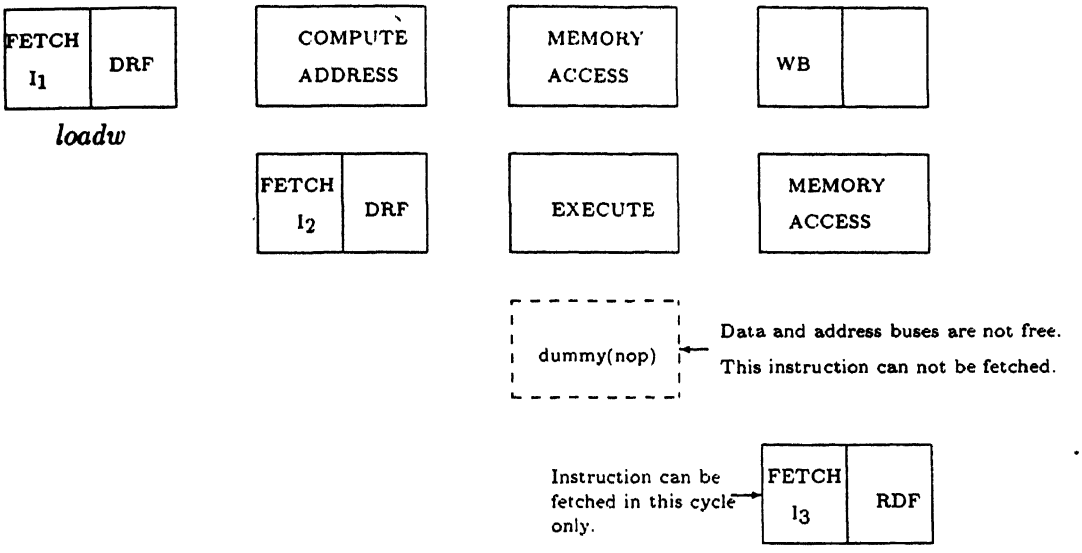


Figure 4.2: Pipeline Suspension During Memory Access.

Delayed Control Transfer

Till this point delayed control transfer(delayed jump/branch) has been used without giving any reason of its inclusion in pipelined architecture. In this section concept of delayed control transfer has been introduced formally. Consider the execution sequence shown in figure 4.3.

During first part of jth cycle instruction I_1 is fetched, which is a control transfer instruction. It is decoded in second part of jth cycle. During cycle (j+1), the target address for control transfer instruction is computed and the necessary boolean condition is also evaluated. The boolean condition is evaluated during the first part of (j+1)th cycle. Along with the execution of I_1, a new instruction I_2 is also fetched and inserted into pipeline for decoding and operand fetch because till the end of (j+1)th cycle the execution of control transfer instruction is not completed (target address has not been computed). At the end of execution phase the target address of control transfer instruction is available and in case of conditional jump type instructions the decision about the successful or unsuccessful jump has been taken. If control is to be transferred then target address of the control transfer instruction is loaded into the PC in the end of (j+1)th cycle and the fetching now can be done from the target of control transfer instruction from cycle (j+2) and onwards.

The instruction I_2, which is already in pipeline, may be flushed out of the pipeline or left inside the pipeline. The first solution is not attractive because CPU has already decoded I_2 and flushing it will waste one cycle. Further flushing the pipeline will require extra control

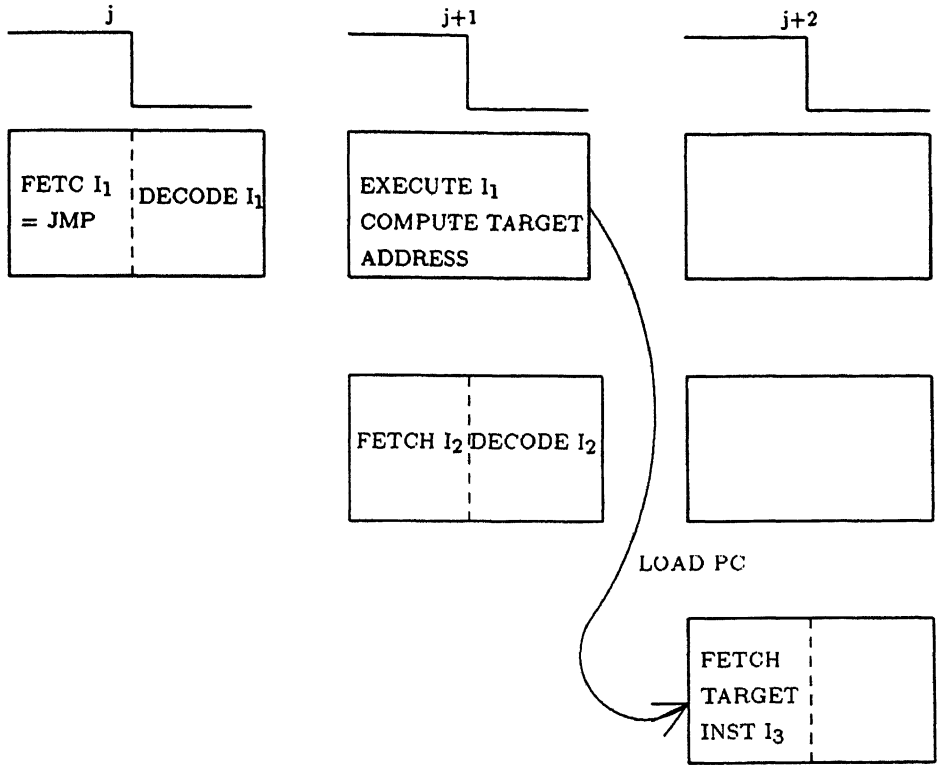


Figure 4.3: Delayed Control Transfer.

and will complicate the flow of pipeline. The iitk-RISC uses the second option because it does not require any other extra hardware and no clock cycle is wasted.

The effect of leaving I_2 as such is that the control transfer is virtually delayed by one cycle hence the name delayed control transfer. The instruction slot I_2 is referred to as the Delay Slot of instruction I_1. The instruction in delay slot will always be executed and compiler can put an instruction in delay slot which does not affect the decision of the control transfer is not going to be effected. Most of the time compiler will be able to find a meaningful instruction suitable for delay slot but occasionally compiler will insert a *nop* instruction there. Empirical results have shown that the compiler is able to fill the delay slot of unconditional control transfer instruction about 90% of the time and delay slot of conditional control transfer instruction for the 40% to 60% of the time [5, 9].

If two or more control transfer instructions are executed in consecutive cycles then the control will be transferred to the target address of the last executed control transfer instruction. However, instruction at the target address of control transfer instruction is also executed in the delay slot of subsequent control transfer instruction. This situation is illustrated in figure 4.4.

```
90 ···              ;finaly control will be here
    ⋮
100 jmp 150         ;I1
124 jmp 200         ;I2, delay slot of I1
128 ···
    ⋮

150 add ···         ;I3; delay slot of I2
154 ⋮

200 jmp 300         ;I4, control will be here after I1 and I2
204 jmp 360         ;I5, delay slot of I4
208 ···
    ⋮

300 jmp 90          ;I6, delay slot of I5
304 ···
```

Figure 4.4: The Effect of Consecutive Jumps.

```
I1: loadw R1,R3,R2
I2: add  R4,R5,R6
I3: sub  R5,R7,R3
I4: sub  R5,....R7
```

Figure 4.5: An Example Requiring Internal Forwarding.

The Internal Forwarding

In pipelined architecture an instruction is executed in every cycle and a new instruction is inserted into the pipeline without waiting for the earlier instruction to finish. Consider the program segment shown in figure 4.5.

In non-pipelined execution there is no problem in executing this code segment but in case of pipelined execution the effect of this program segment will not be as expected because operand registers R_5 and R_3 of I_3 will be fetched from the register file before they have been written. That means, the *sub* instruction will operate on old content of R_5 and R_3. Same is the case with I_4. To get consistent result, in pipelined computer, the result of the

instructions already in pipeline is forwarded to subsequent instructions. This is referred to as Internal Forwarding.

In iitk-RISC the places at which internal forwarding is done are shown in figure 4.1. Data manipulation instructions, data movement instructions, and control transfer instructions require internal forwarding from memory access unit to execution unit and write back unit to execution unit. Load/Store type instructions require internal forwarding from write back unit to memory access unit in addition to above two.

4.2 The iitk-RISC Architecture

The basic block diagram of iitk-RISC is shown in figure 4.6. In subsequent sections every block is explored up to its circuit level design and design parameters are also discussed for the same. The implementation details of all the units discussed here can be found in next chapter.

4.2.1 The Register-File

iitk-RISC has 128 registers of 32 bit each and all registers are organized in a group and as a whole they are referred to as *Register-File*. The register file has two ports for reading and writing operations. The $port_1$ is attached to system Bus A and $port_2$ is attached to system Bus B through register RA and register RB respectively. For reading operation a port is used. Simultaneously two registers can be read as there are two ports available. In writing operation data is read from $port_1$ of register file which is demultiplexed and attached to system BUS D through RD. Registers RA and RB are the output buffer registers whereas RD is input buffer register for register-file.

The register R_0 is hardwired to contain zero and writing into R_0 is allowed without any effect. This is a special purpose register and may be used as the destination of the instruction in which user does not want to change the content of any of the registers, for example to compare two registers a subtract instruction is used with destination of the result of this subtraction instruction as R_0. The read and write operations on the same register simultaneously are avoided by carefully designing the pipeline. Refer to figure 4.1, the operands of an instruction are read during low part of the cycle (clock is low) but the result of an instruction is written only during high part (clock is high) of the cycle. Therefore reading and writing operations on register-file will not clash.

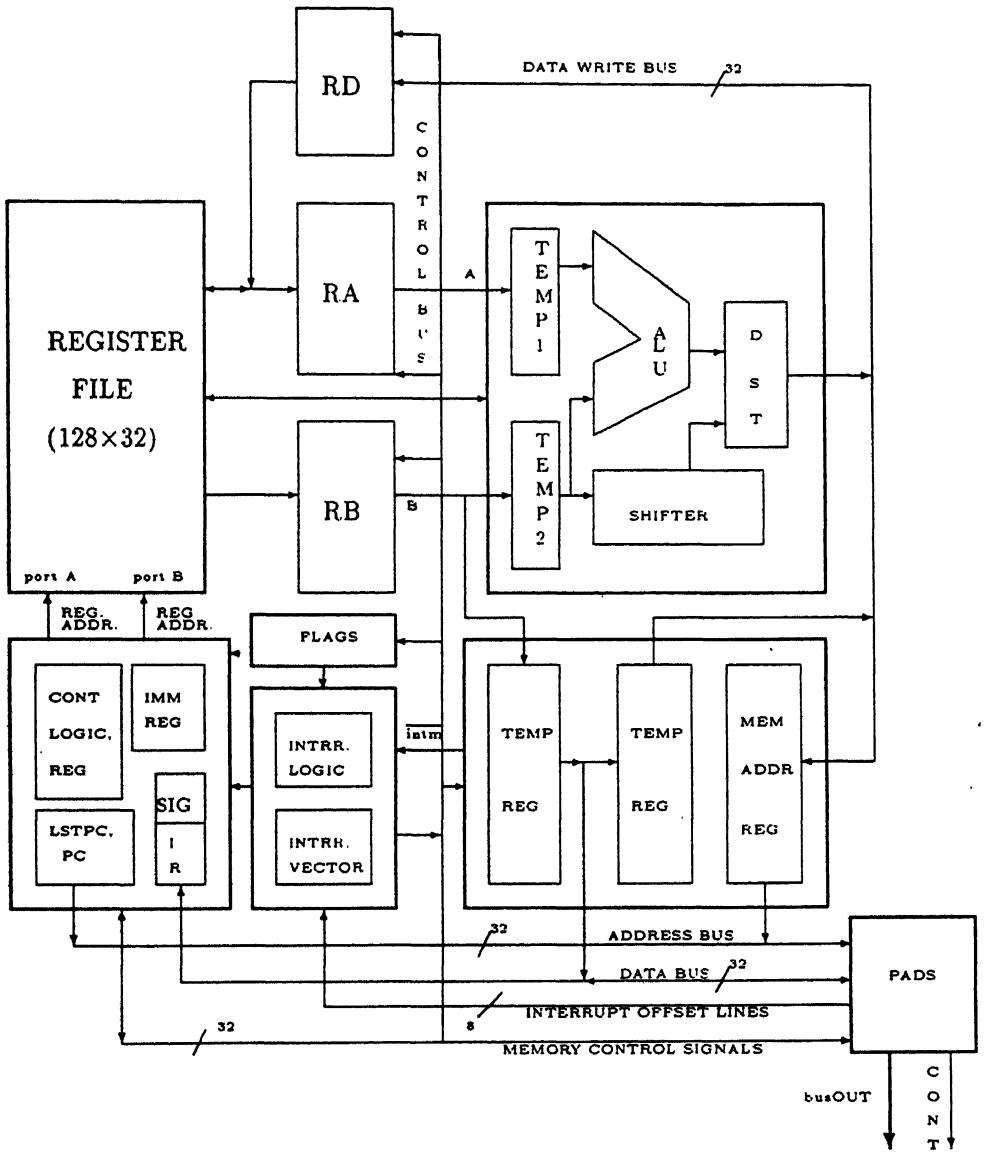


Figure 4.6: The iitk-RISC Block Diagram.

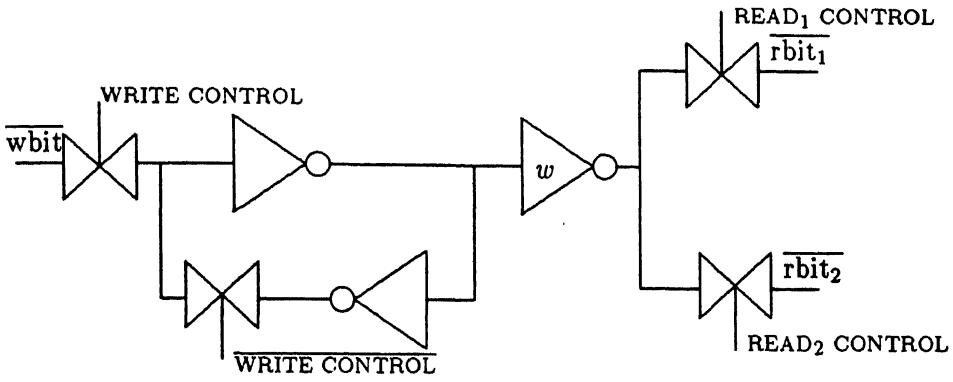


Figure 4.7: A Basic Cell Register.

The Register Cell

A register is nothing but the replication of a basic cell shown in figure 4.7. The cell is a simple CMOS D-latch with two lines for reading and a line for writing. The choice of D-latch over static RAM-cell is critical one. In case of RAM-cell the design of sense amplifier is critical and slight over or under design can affect the data read. The way D-latch is implemented costs almost same area as static RAM-cell.

Register Address Decoder for Register-File

Seven address lines are required to address all 128 registers of register-file. The iitk-RISC uses two register address decoders corresponding to two ports of register-file. Separate decoders are used for two register read operations and the first register address decoder is also used for writing operation. A simple AND-tree decoder is used for designing the decoder. Figure 4.8 shows an AND-tree decoder for 3 address lines.

4.2.2 Execution Unit

The execution unit of iitk-RISC comprises of a 32 bit add/sub unit, a logical unit for 32 bit operations, and a shifter for 0 to 31 bit shift operations. The execution step of iitk-RISC execution pipeline is performed by this unit. It takes two inputs from two temporary registers (It is assumed that input will always be available in these two registers before any operation is started to be performed by this unit) and after performing the operation requested for result is stored in DST-reg. At the end of computation ALU returns either the old value of condition flags or the modified values of flags. If SCC bit of instruction is

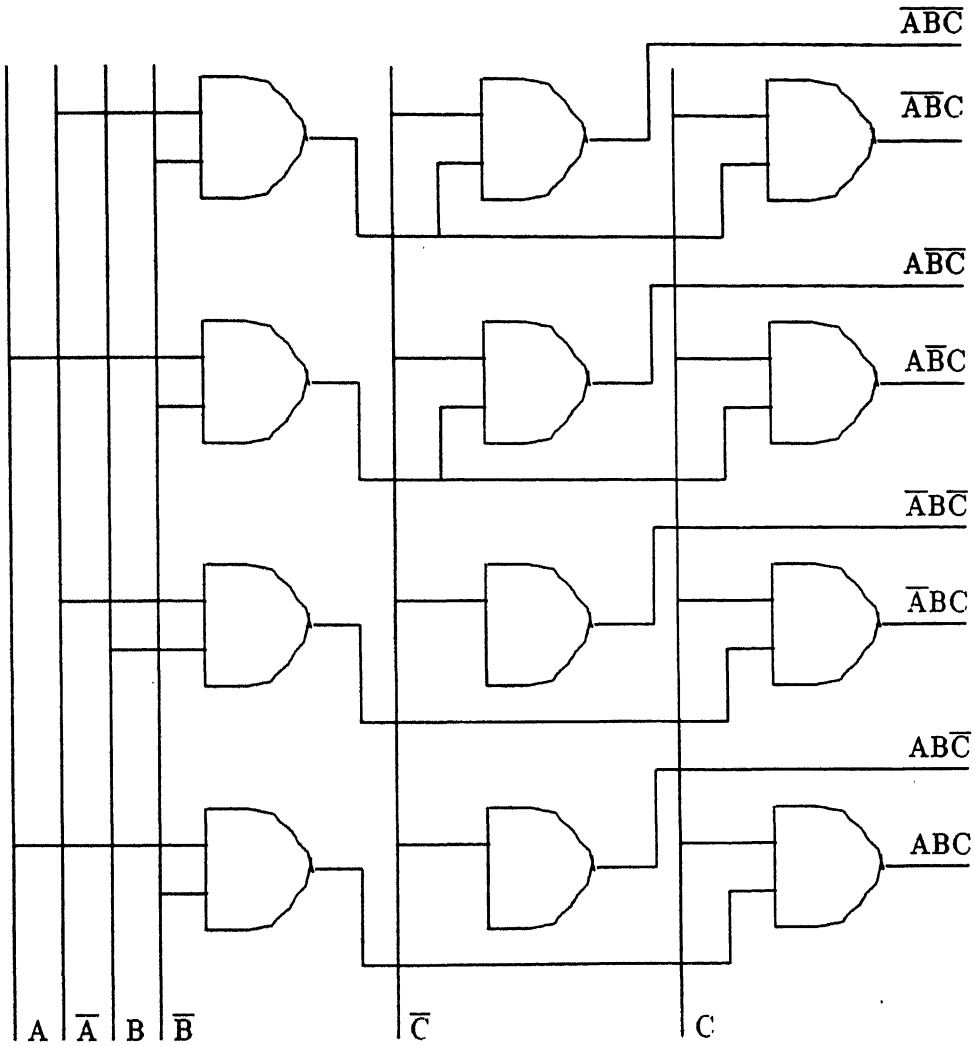


Figure 4.8: An AND-tree Decoder for Three Lines.

OFF or instruction is not permitted to modify the flag (even SCC bit is 1) then old content of condition flags is returned otherwise modified content is returned.

Adder/Subtractor

We have chosen a carry-lookahead adder because the addition and subtraction operations are to be done in a single cycle. A carry lookahead adder consists of a carry generation block which generates all the carry bits (or block of these) in parallel and an add block which performs the actual addition. We have used a modified version of normal carry-lookahead adder because of reduced complexity and implementation reasons.

Consider the two numbers A and B , and a number S such that $S = A + B$. Let A_i, B_i, S_i, and C_i be the bit i of number A, bit i of number B, bit i of sum S, and carry out

of sum S_i respectively. The carry out of stage i may be expressed as

$$C_i = G_i + P_i.C_{i-1}$$

where $P_i = A_i \oplus B_i$, carry propagation signal.

$G_i = A_i.B_i$,carry generation signal.

then $S_i = A_i \oplus B_i$

We can define a new operator o which has the following function:

$$(g,p) \circ (g',p') = (g + (p.g'), p.p')$$

where g, p, g', p' are boolean variables. The carry signal can be determined by

$$C_i = G_i$$

$$where [G_i, P_i] = \begin{cases} (g_1.p_1) \circ (g_0, p_0) & if\, i = 1 \\ (g_i, p_i) \cdots \circ \cdots (G_{i-1}, P_{i-1}) & if 2 \leq i \leq n \end{cases}$$

The operator o is associative and allows the processing elements to be embedded in a binary tree [11]. Such a carry generation block is shown in figure 4.9 for 8 bit numbers and an input carry C_0. The adding block consists of XOR gates.

Other details can be found in [11]. The iitk-RISC has two 16 bit blocks of the adder discussed above. The adder is appended with an XOR gate for each bit to get the one's complement of second operand in case of subtraction operation and normal addition is performed with input carry C_0 is set to 1. In case of addition operation XOR gates pass the bits of second operands as such.

Logical-Unit

The logical-unit of iitk-RISC is capable of performing bitwise AND, OR, and XOR of two 32 bit numbers and bitwise inversion of a 32 bit number. The logical unit is designed by replication of the basic cell shown in figure 4.2.2.

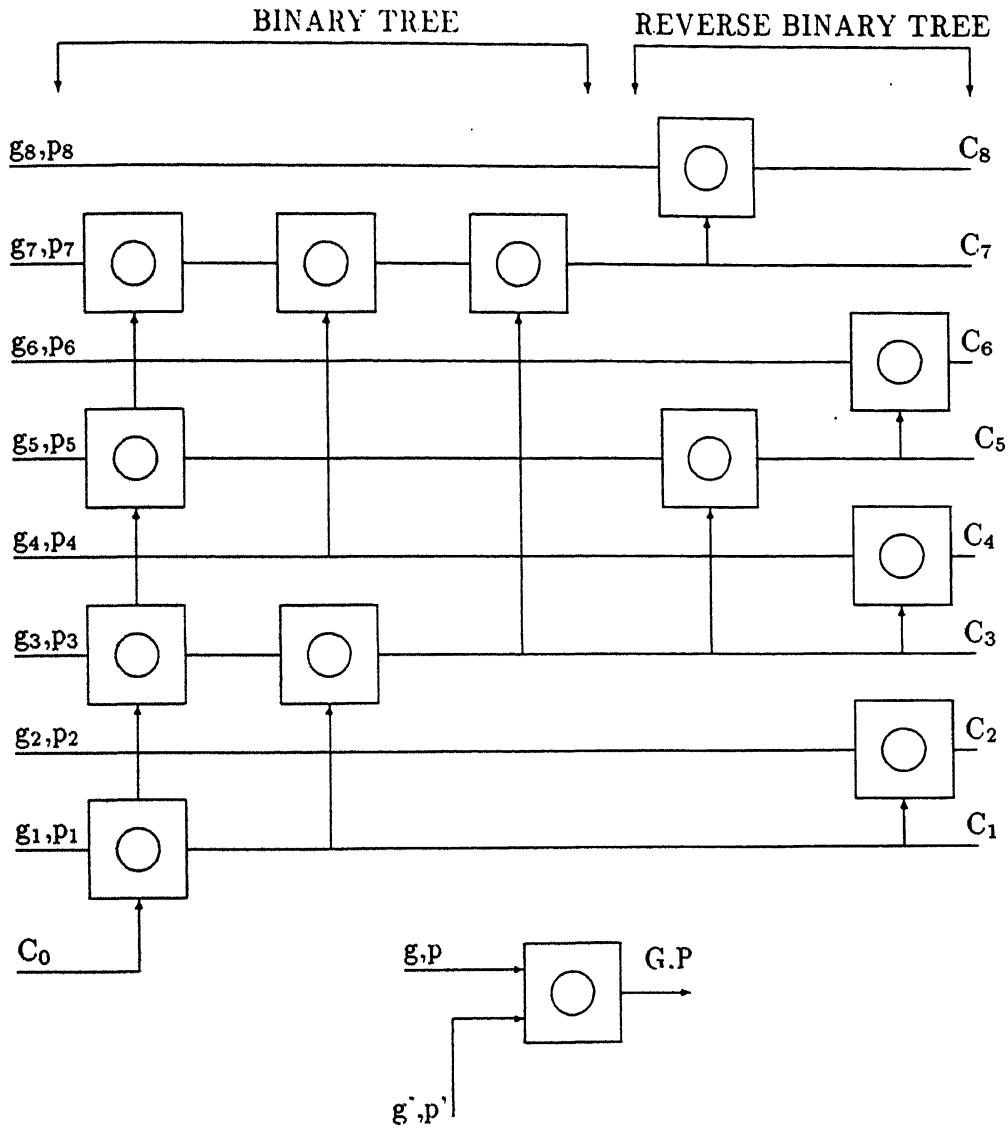


Figure 4.9: A Binary Carry Lookahead Adder.

Shifter

A barrel shifter is used to shift 32 bit number read from BUS A by the amount given by the shift-count (shift-count<32) read from BUS B. To get the shift-count five low order bits of $TEMP_2$ register are used. The design of barrel shifter consist of a basic shift block, a shift count decoder, and some extra hardware to make the basic block capable of doing all type of shift operations. The basic shift block is a 32×32 matrix of pass transistors and it is capable of doing right shift operation. The basic shift block has two input planes and a output plane. The control signals are fed in vertical direction, inputs(data to be shifted) are given diagonally, and outputs(shifted data) are taken out in horizontal plane. The output

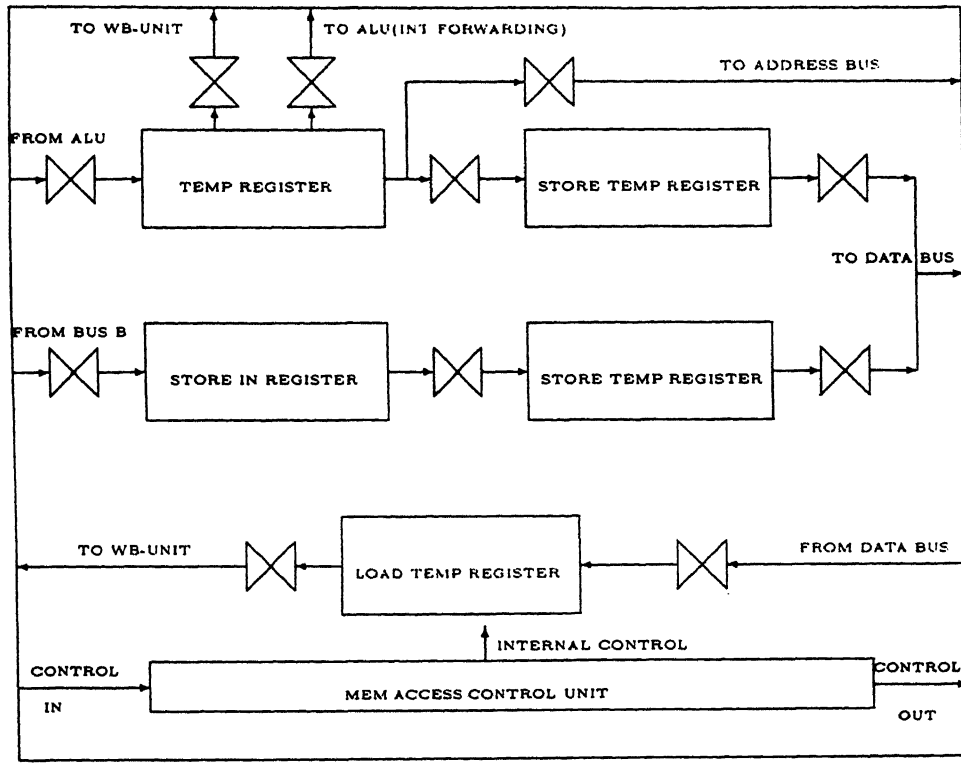


Figure 4.11: The Memory Access Unit of iitk-RISC.

of store type instruction it outputs the data together with the information about the width and the memory banks to be enabled as discussed in Chapter 3. The basic architecture of memory access unit is shown in figure 4.11 and the timing diagrams for read and write cycle are shown in figure 4.12. The *request* signal shown in read and write cycles is used internally and shown here for clarity.

4.2.4 Write Back Unit

The basic architecture of write back unit is shown in figure 4.13. The write back unit takes the input from memory access unit during low part of cycle and writes the result into register-file or flag register during high part of cycle.

4.2.5 Interrupt Unit

The block diagram of interrupt unit is shown in figure 4.14. The interrupt request may come from an external device or internal unit. Internal request can come from control unit for illegal instructions or memory access unit for address violations. The following actions are taken by the interrupt unit whenever it receives an interrupt request on one of the

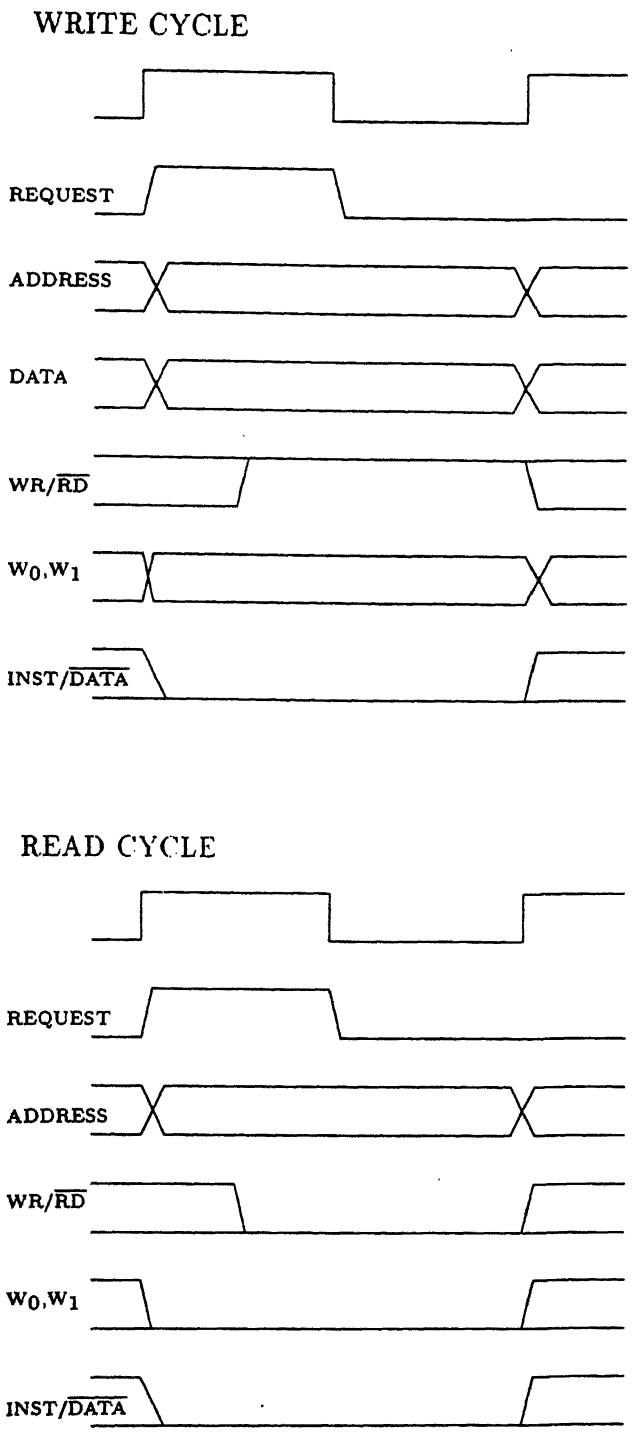


Figure 4.12: The Read and Write Memory Cycles.

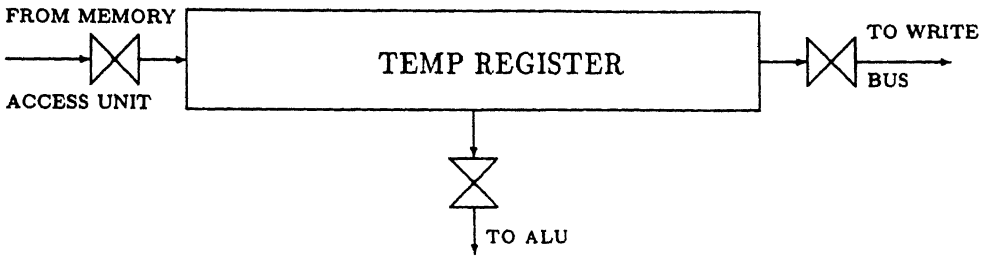


Figure 4.13: The Write Back Unit of iitk-RISC.

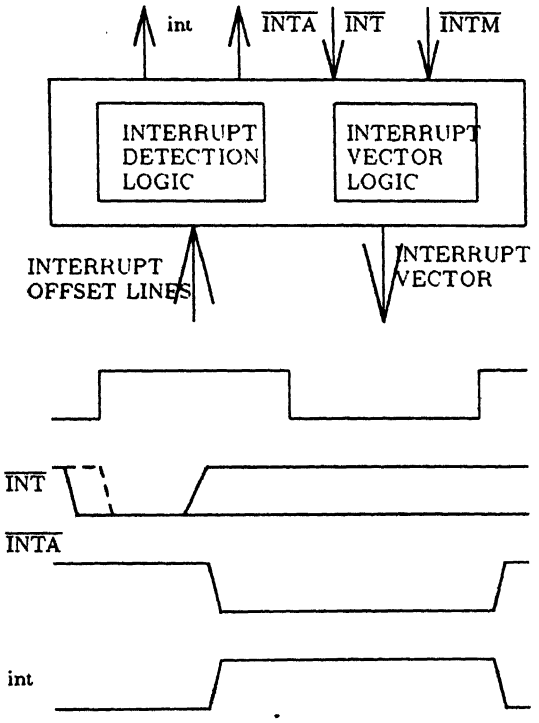


Figure 4.14: Interrupt Unit and Interrupt Cycle.

interrupt lines:

- On *int* signal: [P]→T, [signature register]→LSTPC, P←1.
- Further fetching of instruction is suspended and a *nop* instruction is inserted into the pipeline.
- 8 bit of interrupt offset is generated either internally or after reading the 8 interrupt lines and using this interrupt offset interrupt vector is calculated.
- PC is loaded with the interrupt vector just computed, and fetching of instruction is resumed from that address.

Interrupt line $\overline{INT}$ is checked in the begining of every cycle for the presence of interrupt request from external device but other internal interrupts are processed whenever detected. The interrupt due to memory address violation arises during the high part of cycle(clock is high). However, interrupt due to illegal instruction arises during low part of cycle(clock is low). When an interrupt request from external device or memory adders violation is detected an interrupt cycle is initiated. In case of interrupt request due to illegal instruction a short length interrupt cycle is initiated. This short length cycle takes half of the cycle to complete. Interrupt cycle once started can not be stopped and no further request from external device is entertained. The interrupt cycle completes at the end of current cycle. In figure 4.14 an interrupt cycle is shown. $\overline{INTA}$ is the acknowledgement signal for $\overline{INT}$. External interrupts are maskable and can be disabled by clearing the I flag. The two internal interrupts are unmaskable but interrupt due to invalid instruction has low priority. Interrupt due to memory address violation has high priority. The interrupt vector is computed as : interrupt-vector=FFFFF800h +offset×8h. The interrupt table of iitk-RISC starts from memory address FFFFF800h and extends up to FFFFFFFFh. Two consecutive full words are reserved at each interrupt vector location as the first instruction here is an unconditional jump and due to delayed control transfer the next instruction is also executed. The interrupt vector locations FFFFF800h and FFFFF808h are used for memory address violation and illegal instruction respectively.

4.2.6 Control Unit of iitk-RISC

The structure of control unit is shown in figure 4.15. Control logic is divided into parts and almost all the units of iitk-RISC have a small control section. However, most of the control logic is inside control unit. This has reduced the number of control lines considerably. In this section fetching and execution process of instruction in iitk-RISC is discussed.

Fetching a Instruction

The part of control unit responsible for fetching instruction includes the Program Counter (PC), the instruction register, and the signature register. The PC is a 32 bit counter and bits<1:0> of the address generated by it are always 00. This is because all iitk-RISC instructions are aligned at word boundaries and memory is organized in byte banks. The fetch control first identifies whether address and data buses are free or not. Buses may be

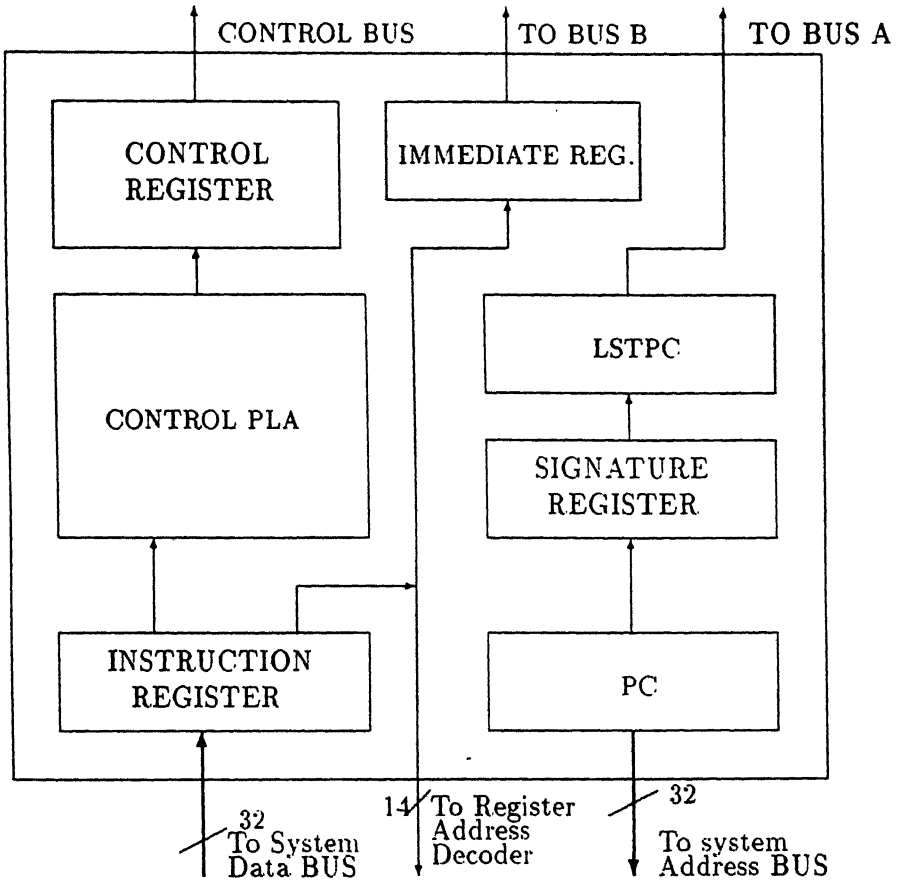


Figure 4.15: Control Unit of iitk-RISC.

occupied by memory access unit for memory access operation, which is given first priority because a pipeline stage can not be delayed. To see whether buses are free or not, fetching control accesses the busy bit of memory access unit. If busy bit is set the buses are not free otherwise buses are assumed to be free. If buses are not free then fetch cycle is not initiated and a *nop* is inserted into the pipeline. In other case, when buses are found to be free, a fetch cycle is initiated. The iitk-RISC fetch cycle is shown in figure 4.16.

During the fetch cycle the content of PC is released onto the address bus and signal is sent to inform the memory that a fetch cycle is in progress. The content of PC is pushed into the signature register and the data bus is read after sending the $\overline{RD}$ signal. The instruction is send for decoding and operand fetch. If *int* signal is received by the control unit from interrupt handling unit the fetch cycle just initiated is suspended and *nop* is pushed into the pipeline.

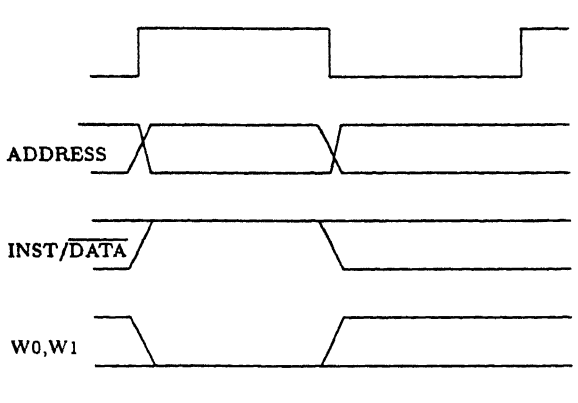


Figure 4.16: A Fetch Cycle of iitk-RISC.

Control Signals Generation

The control signal generation part of control unit consists of instruction register, control PLA, and control register. The instruction is fetched from memory into instruction register. The instruction register is designed with simple D-latch with clear signal. Previously at several places we have mentioned that on occurrence of some particular signal a *nop* is pushed into the pipeline. The *nop* is pushed into pipeline by clearing the instruction register. We have kept the opcode of *nop* 00000000h intentionally for the same reason. Signature register works like an identification tag to a instruction register. The signature register has the memory address of a instruction that is in instruction register and is loaded during the instruction fetch. The signature of the instruction is used by several instructions. These include all the PC-Relative control transfer instructions. If control unit gets *int* signal from the interrupt unit, then the LSTPC is loaded with the content of signature register.

Following is the list of the control signals that are required to manage the instruction execution in iitk-RISC.

- Control signals required by control unit.

  1. C_0: A valid instruction.
  2. C_1: Load P flag with the content of T flag.

- Control signals required by execution unit.

  1. A_{-1},A_0: Execution unit input source for first operand. 00:RA, 01:Signature register, 10:flag register, 11:LSTPC.

2. A_1,A_2: Execution unit input source for second operand. 00:RB, 01:immediate register, 10:Z-register.
3. A_3,A_4: Execution unit subunit activation code. 01:add/sub unit, 10:logic unit, 11:barrel shifter.
4. A_5, A_6: Execution unit operation code.
   00:add,and,shr; 01:sub,or,shl; 10:adc,xor,sar; 11:sbb,not.
5. A_7: Flag control line. 0:condition flags will not be changed, 1:Condition flags will be changed.
6. A_8: Send PC content to memory access unit in low part of cycle.

- Control signals required by Memory Access Unit.

1. M_0: Enable/$\overline{\text{Disable}}$ unit.
2. M_1: Operation type. 0:load, 1:store.
3. M_2: Signed/Unsigned data. 0:unsigned, 1:signed.
4. M_3,M_4: Data width. 01:byte, 10:half word, 11:full word.
5. M_5: Data currently with execution unit will be written in register file. Used for internal forwarding mechanism.
6. M_6: Send data received from execution unit to write back unit.
7. M_7: Send data accessed from memory to write back unit.
8. M_8: Send PC content received from execution unit to write back unit.

- Control signals required by write back unit

1. W_0: Enable/$\overline{\text{Disable}}$ unit.
2. W_1: Data currently with memory access unit will be written in register file. Used for internal forwarding mechanism.
3. M_2: Destination of data. 1:register-file, 0:flag register.

If internal forwarding is done, then some of the signals lose their meaning. These signals are (A_{-1},A_0) and (A_1,A_2). If internal forwarding is to be done for first operand then A_{-1} and A_0 loose their meaning. If internal forwarding is to be done for second operand then A_1 and A_2 loose their meaning. In case internal forwarding is to be done

for both operands then (A_{-1},A_0) and (A_1,A_2) all loose their meaning. Signals M_5 and W_1 are used to control internal forwarding. If one or both register operands of an instruction are also the destination register of the previous instructions and are not written back yet as the instruction has not passed through write back unit, then modified data is internally forwarded and not fetched from register-file. The internal forwarding is done in the following steps:

a. First the location for internal forwarding is identified as follows:

   - The instruction whose destination matches with instruction currently in decoding process is identified, say I. If there are two such instructions inside the pipeline then the instruction nearer to the entry end is taken as instruction I because it is the instruction whose result will be the final content of the register.
   - The location of instruction I is identified, say L.
   - The location of instruction I in pipeline when internal forwarding will be done to provide the correct data to the current instruction is identified. It will be $L+1$ when internal forwarding is to be done from write back unit to execution unit and $L+2$ when it is to be done from write back unit to memory access unit.

b. After identification of the place of the instruction I control signals M_5 or W_1 are checked of the instruction I. The control signal M_5 is checked if instruction I currently is in execution phase of execution cycle and W_1 is checked when instruction I is in memory access phase of execution cycle. After this internal forwarding control signals are generated using above information and the information about the equality of source register of current instruction and destination register of instruction I. These signals will be used to forward result on BUS A, BUS B, or on Data bus(store type instruction).

All control signals are generated during the decode phase of execution cycle. A PLA is used to generate all the control signals. The generated control signals are stored in a control register and the signals are issued to each stage of the the pipeline in the order of execution cycle.

Control PLA and Control Register

We have used a CMOS NOR-NOR dynamic PLA for designing iitk-RISC control logic. The CMOS NOR-NOR PLA is the fastest among all CMOS PLAs because of parallel connected NMOS-transistor in AND and OR planes.

There are 14 inputs to the PLA and 25 outputs are generated. Inputs are opcode(7), SCC bit(1), field identification bits(2), P flag(1), Conditional flags(4). The opcode, SCC bit and field identification bits are taken from the instruction to be decoded and currently in IR.

Output control signals, destination register address, and information about internal forwarding are stored in control register. Control register is a three stage register. The information flows from stage 1 to stage 2 to stage 3. Each stage is made of master-slave CMOS D-latch with clearing facility. The first stage has 35 cells. These cells are used for storing destination register address(7), internal forwarding control signals for internal forwarding from MEM stage to EXEC stage(2), internal forwarding control signals for internal forwarding from WB stage to EXEC stage(2), internal forwarding control signal for internal forwarding from WB stage to MEM stage(1), control signals for write back unit(3), control signals for memory access unit(9), and control signals for execution unit(11). The second stage has 20 cells. These cells are used for storing destination register address(7), internal forwarding control signal for internal forwarding from WB stage to MEM stage(1), control signals for write back unit(3), and control signals for memory access unit(9). The third and final stage has 10 cells. The cells are used to store destination register address(7), and control signals for write back unit(3). The first stage issues the control signals to execution unit, second stages issues signals to memory access unit while the third stage issues the control signals to write back unit. The internal forwarding control signals are sent to the unit whose result is to be internally forwarded. The control signals remain active throughout the cycle but are used by their controlled unit either in low or high part of cycle. The control register of iitk-RISC is shown in figure 4.17.

4.3 Evaluation of iitk-RISC Architecture

Here we have listed down some of the important characteristics of the iitk-RISC to conclude our discussion on the architecture of iitk-RISC.

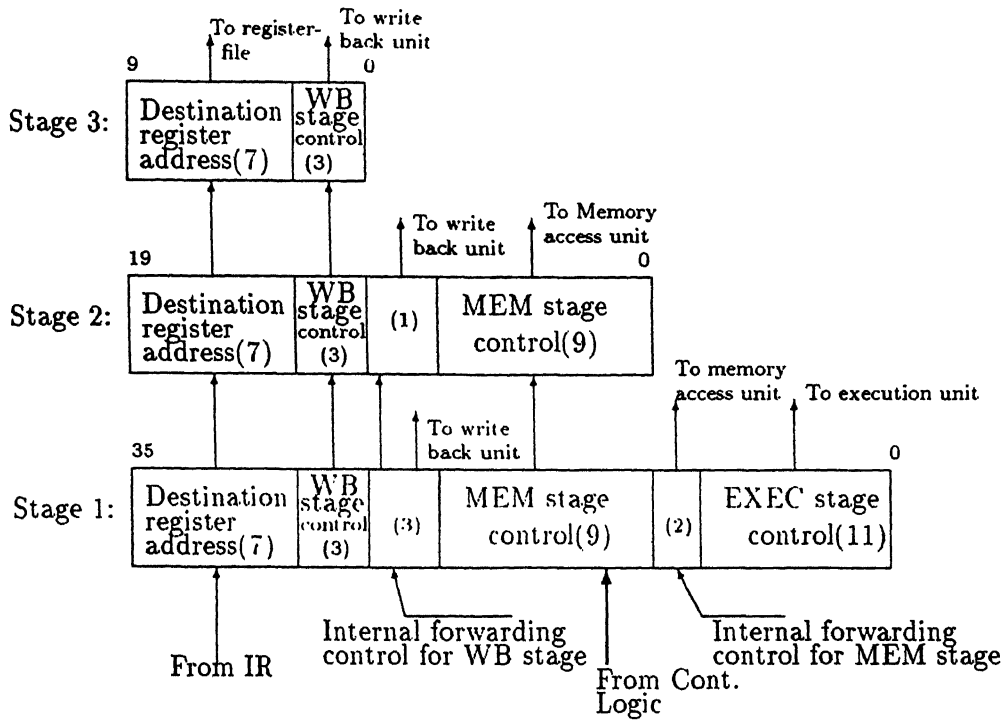


Figure 4.17: The control register of iitk-RISC.

- A simple architecture.
- A 4-stage regular pipeline.
- Simple control mechanism.
- Balanced read and write cycle.
- Fast fetch mechanism.
- Fetching of operands in parallel with decoding.

Chapter 5

The iitk-RISC Design and Layout

This chapter deals with the micro architecture of iitk-RISC. After a detailed description of the data path, layout design issues for different units are discussed. Actual layout for different units are also given.

5.1 The iitk-RISC Data Path Design

For compact layout the data path should be designed carefully. The general form of data path is a direct consequence of the pipeline scheme employed and the instruction set. The basic form of a data path is that a gate is driving some other gates through a resistive and capacitive path. Several designs of data path are possible. These include a precharged path, self amplified path, a static data path. The precharged path requires a multiphase clocking scheme. The data path are precharged to high logic but here discharging of data path is a critical issue. That means the NMOS transistor of the gate driving the data path should be designed carefully. Otherwise in fast operation on data path will not be possible because the driver will not be able to transfer 0 logic. Self amplified or self adjusted data path are suitable for low speed high capacitive and resistive data path. In this scheme pull ups and pull downs are used. The static scheme is very critical from the designer point of view. In this scheme the driver, generally a inverter, is designed in such a manner so that it is able to charge and discharge the data path in specified time. Generally data path is divided into segment and a driver is used for driving every segment to reduce the loading on driver. There is always a trade off between driver size and the segment length a driver can drive. To drive a long segment with a reasonable switching time a fat gate will be required. It is generally used for high speed operations. In iitk-RISC we have used static data path scheme. Though it costs us much in terms of area but works satisfactorily. The segment of

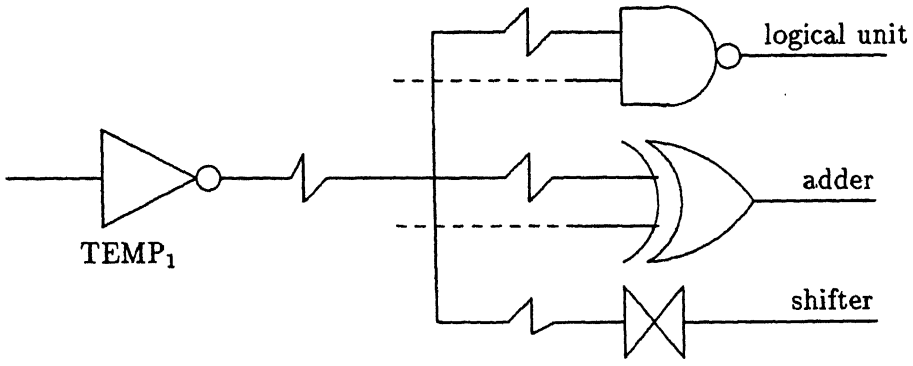


Figure 5.1: A Typical iitk-RISC Data Path Segment.

data path for time critical signal is kept short. A typical iitk-RISC Data path segment is shown in figure 5.1, this data segment drives the input side of execution unit.

5.2 The iitk-RISC Data Path

Figure 5.2 presents the iitk-RISC data path which consists of the following:

- *Register-file:* 128 registers of 32 bit each, with its dual-port address decoder and with latches RA, RB, RD. R_0 is hardwired to contain 0.
- *Flag register:* A 32 bit register. It consist of conditional flags(S,Z,V,C) and flags for system management(I,P,T).
- *Immediate Register:* A 32 bit register and holds the immediate fields of instruction after sign extension to 32 bit data/offset.
- *Z-register:* A 32 bit register hardwired to contain 0 and used as second operand of some instructions internally.
- *$TEMP_1$, $TEMP_2$:* 32 bit temporary registers used as the input registers for execution unit.
- *DST register:* A 32 bit temporary register and used to store the result of execution unit.
- *Execution Unit:* The execution unit take input form $TEMP_1$ and $TEMP_2$ and output is sent to *DST register*. The execution unit comprises of ALU and a shifter. The shifter uses 5 LSBs of $TEMP_2$ as shift-count.

- *IR:* It holds the instruction fetched during fetch cycle. Opcode field of instruction is decoded and other fields are used to fill immediate register and/or used to fetch the registers. Destination register address is send to control section.
- *Signature:* A 32 bit register is used to store the memory address of the instruction currently in *IR*.
- *PC:* The program counter, which holds the address of the instruction to be fetched in current cycle.
- *LSTPC:* The Last-PC register, which holds the content of *Signature* when *int* signal is active.
- *Memory Access Unit:* This unit is used to access the memory in case of *load* and *store* type instructions. It reads effective address from *DST* data to store from RB and accessed data is sent to RD.
- *BUS A, BUS B:* The register-file read buses. Data traverse from RA, RB to other units.
- *BUS D:* The register-file write bus. Data traverse from other units to RD.
- *SYSTEM ADDRESS BUS:* The address of memory during data access or instruction fetch is send across this bus.
- *SYSTEM DATA BUS:* The data or instruction are fetched over this bus.
- *Interrupt Offset Lines:* Interrupt offset is read on the request of external interrupt.
- *busOUT:* The off-chip data and address bus.
- *CONTbusOUT:* The off-chip control signal bus. The off-chip control signals include WR/*overline*RD, $\overline{\text{INT}}$, $\overline{\text{INT}}$, W_0,W_1, and interrupt offset lines.

5.2.1 Paths Followed for Instruction Execution

In pipelined architecture instruction passes through all the stage of the pipeline during the instruction execution. The result of the stages of the pipeline is stored or buffered in intermediate latches. Data is forwarded from the output latch of one stage to input of the next stage. As discussed earlier the iitk-RISC pipeline is a 4 stage pipeline. During the

course of execution of instruction data followed some specified path to pass through the pipeline. Data path of iitk-RISC is presented in figure 5.2. There are a few activities that may be going on in the data path during each cycle.

- The two source operands of instruction are routed to the execution unit.
- The output of execution unit or of the content of signature register is routed to memory access unit or to the PC.
- The output of memory access unit is routed to write back unit or(and) external memory interface. The output of PC is routed to write back unit in case of *call* instruction.
- The output of write back unit routed to register-file or flag register.

The two sources, S_1 and S_2, are routed on different buses or data path. Source S_1 follows the BUS A and it can be any of the data sources attached to BUS A. These sources are RA, Signature register, flag register, LSTPC or internally forwarded result of the instruction already in pipeline. Source S_2 follows the BUS B and it can be any of the data sources attached to BUS B. These sources are RB, immediate register, Z-register, or internally forwarded result of the instruction already in pipeline. The particular choice of S_1 and S_2 depends on the instruction. For example, *getlpc* instruction requires LSTPC as S_1 and Z-register as S_2. The two sources are latched in the input registers $TEMP_1$ and $TEMP_2$ of execution unit respectively. Which are used as the input for all binary operations whereas $TEMP_1$ is the input source of unary operation *inv*. The result of execution unit is forwarded to the output register of execution unit.

The output of execution unit, DST register, is routed to memory access unit or to the program counter. if the instruction is control transfer instruction then the DST register is routed to PC otherwise the content of DST register is forwarded to memory access unit. The other input source to the memory access unit is register RB content delayed by 2 clock cycles if instruction is store type. If instruction is load type then the other source is memory access interface or system DATA BUS.

The output of memory access unit is routed to write back unit or(and) memory access interface. If instruction is store type then output of memory access unit is forwarded only to memory access interface. If instruction is load type the effective address over system

ADDRESS BUS to memory access interface and accessed data is sent to write back unit. In other cases the output of memory access unit is forwarded to write back unit. If internal forwarding is required then output is also forwarded to the input latch(es) of the execution unit. The output of memory access unit is written into register file or flag register. Some instruction do not have any destination field in these cases write back unit do not forward its output to any of the unit. The output of memory access unit may be internally forwarded to the input latch(es) of execution unit or(and) memory access unit. Internal forwarding is discussed in chapter 4.

5.3 The Design Issues and Layouts

In this section the design issues of iitk-RISC layout are discussed. The discussion is depended on the discussion of previous chapter. We have included the layouts of different units and building blocks of these units. The simulation results of some time critical units are also given. The simulation in most of the cases is done with SLS(Switch Level Simulator) [4]. However, some critical parts are simulated with SPICE [10].

The layout of iitk-RISC is illustrated in figure 5.3. The floor plan is overlaid for easy understanding. In the design of different units some of the registers are duplicated to save the space used by metal wires.

5.3.1 The Register-file

In iitk-RISC register-file occupies 39.32% of the whole designed area. The register file is divided into two banks of 64 registers each to make the layout suitable for connecting with other units and to optimize the access time. The critical delay path of register file is shown in figure 5.4.

If read signal is enable then the decoded signal is applied to selected register. The decoded signal will charge 32 output pass transistor(decoder line) of jth register to enable the latched data in jth register. This data follows the output lines(bit lines) of the register-file. The following discussion is given for only one of the decoder lines required to enable NMOS transistor of a transmission gate.

Let t_{dt} and t_{bt} be the delay associated with decoder line and bit line respectively, that is at least t_{dt} will be required to enable the decoder line and for bit line this time is t_{bt}. The access time of a particular register will be $t_a = t_{dt} + t_{bt}$. In case of the design of register-file

t_{dt} and t_{bt} are the two design parameter. The delay t_{dt} depends on size of inverter IV_{dj} and the decoder. However, the delay due to decoder can not be changed very much. The delay t_{bt} depends on the size of inverter IV_{bj} and pass transistor P_{bj}. However, effect of P_{bj} is not much because there are two transistor(a transmission gate) in parallel to charge the bit line. For low access time t_{dj} and t_{bj} should be reduced. In our design the values of t_{dj} and t_{bj} are 7ns and 13ns respectively. We have got these values by careful design of the two critical inverters and by reducing the length of poly lines. The decoder lines are taken as first metal whereas bit lines are taken as second metal. The bit lines are amplified at the output of register-file to avoid further degradation in bit quality.

The layout for register-file together with one register cell is shown in figure 5.5.

5.3.2 Execution Unit

The execution unit of iitk-RISC is most time critical unit because the longest delay path is inside in this unit and is shown in figure 5.6.

The length of this path is approximately 45 gates. It starts from the output of TEMP register and passes through XOR gate, lower carry generation block, upper carry generation block, add block and zero detector(zero detector finds whether result of execution unit is 0 or not). That is Z flag is the last signal that will be available from execution unit. The worst case delay for this path is approximately 51ns.

The layout for execution unit is illustrated in figure 5.7.

5.3.3 Memory Access Unit

The memory access unit of iitk-RISC is designed keeping in mind that no instruction cache is provided. Due to this a full cycle is used for memory access. For laying compact layout the memory access unit is not located at one place and some of its registers are duplicated to avoid long wires. The internal forwarding buses for internal forwarding from any other unit to execution unit are kept with execution unit. This way we have avoided the long run of wires and duplication of data paths. For internal forwarding from write back unit to memory access unit a duplicate register is used inside the memory access unit. The duplication of register at most of the places is done for easy implementation of internal forwarding mechanism. The layout for main part of memory access unit is shown in figure 5.8.

5.3.4 Write Back Unit

This is the simplest unit in iitk-RISC and required least design effort. It consist of two input register corresponding to two sources and a output register with two ports corresponding to two destinations of final result. If result is to be written into any one of the destination then a write signal is sent to that unit and corresponding output port is also enabled. The layout of this unit is given in figure 5.9.

5.3.5 The iitk-RISC Control Unit

In this section we discuss the organization of iitk-RISC and the design issues in control logic design. The control logic is designed with a CMOS NOR-NOR dynamic PLA. The main issue in designing the control unit of iitk-RISC is to minimize the number of control signals.

The organization of iitk-RISC control path is given in figure 5.10. Decoding of instruction in iitk-RISC is easy because of the simple instruction format with fixed fields of fixed size and positions. The decoding is further simplified because of the small number of addressing modes supported in iitk-RISC. The assumption that all possible operands are available with an instruction allow iitk-RISC to fetch all the operands in parallel with decode of instruction and require no control for it.

The instruction is fetched in IF phase. The opcode field together with condition flags, privilege flag and the fields of instruction specifying the addressing mode used are decoded by the control logic. At the end of decoding all the 25 control signals are pushed into the control register. Control signals for internal forwarding are also generated in parallel with decoding and pushed into control register. These internal forwarding signals may or may not enable a particular type of internal forwarding. The control register, a 3 stage register, issues control signals to all the units of iitk-RISC.

The control logic is designed to enable fast decoding of instructions and is divided into three parts. Each part is implemented using a CMOS NOR-NOR dynamic PLA. We observed that condition flags are updated only at the end of cycle and the control signals affected by these flags are required in the end of cycle. Due to this, these control signals can not be generate during the decoded phase. Keeping in view of this fact the control signals used to load the PC are generated by separate decoding logic and it is one of the three parts mentioned earlier. The other two parts are used to generate control signals for control unit and execution unit, and memory access unit and write back unit respectively. This

way we have reduced the total number of minterm required to implement the control logic. Further reduction in total number of minterm is achieved by multivalued optimization using **espresso** [3]. A total of 72 minterms are implemented in all the three PLAs to realize the control logic of iitk-RISC. The layout of control unit is given in figure 5.11.

5.4 Conclusion

The iitk-RISC is implemented in c3tu process with minimum feature size 1.6μm. Total size of the processor implemented is 7.85 x 7.25mm (56.95mm^2) without pads. This can be easily integrated in a wafer. Total number of pins required are 82.

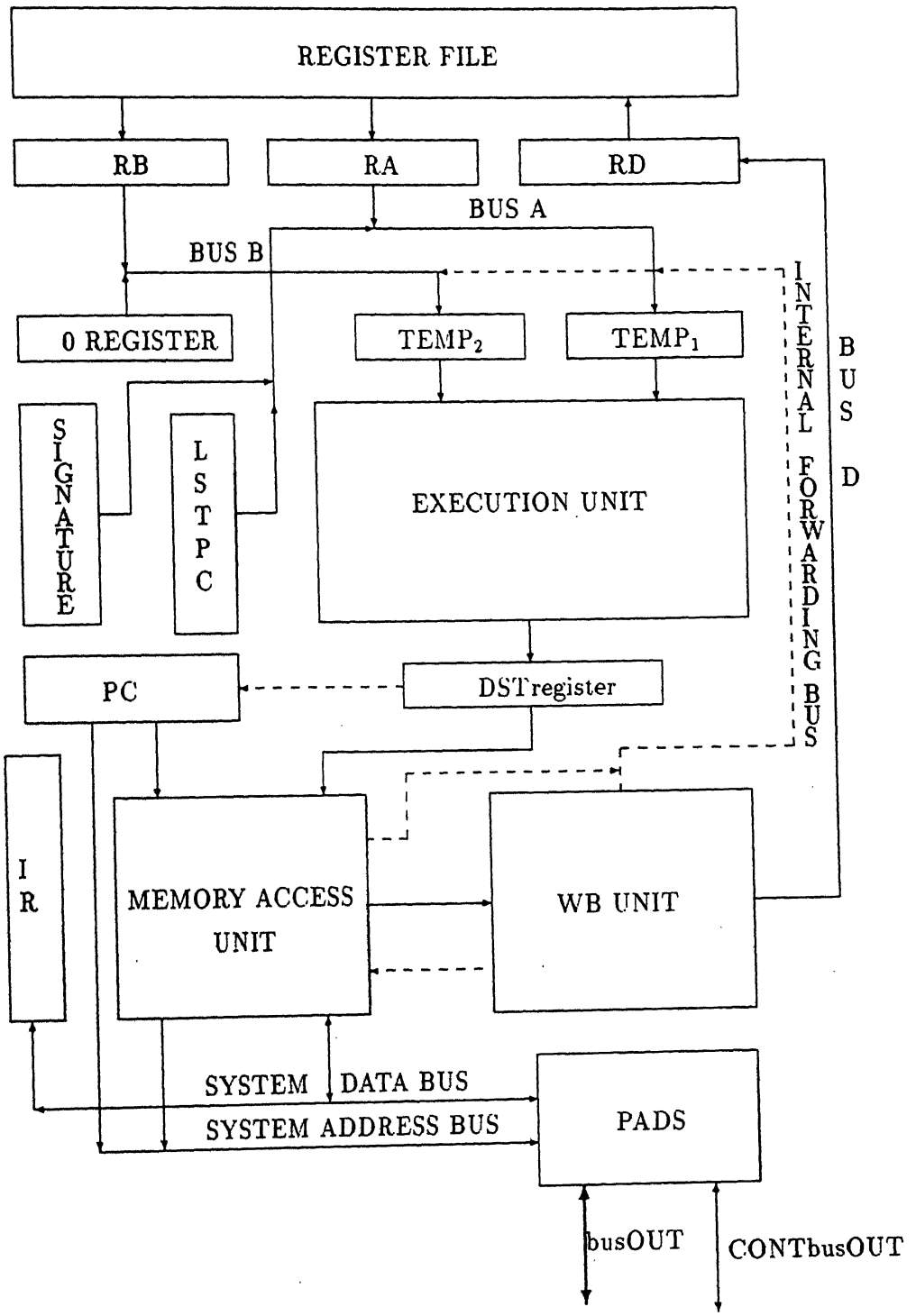
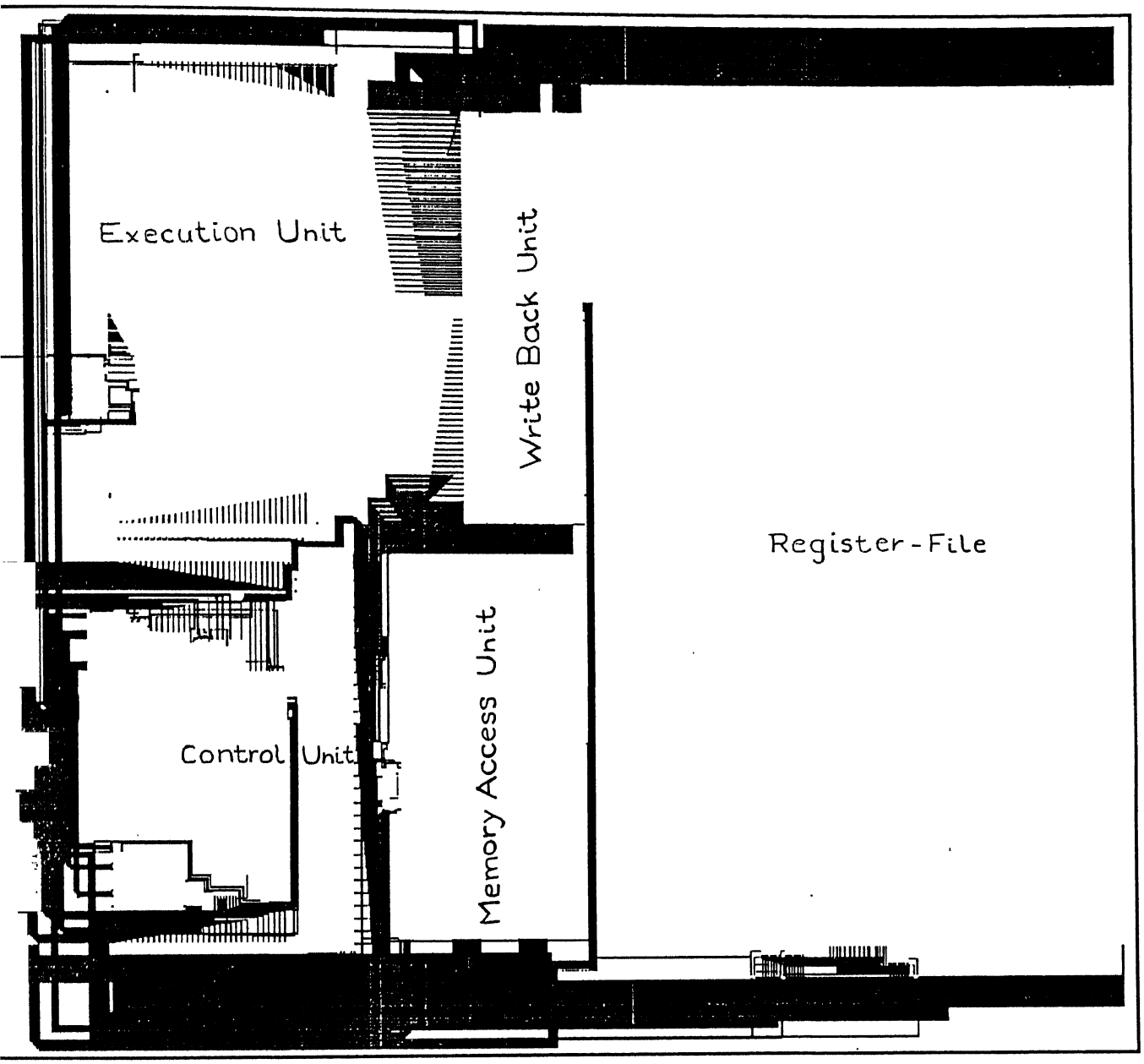


Figure 5.2: Data Path of iitk-RISC.



Size:7.85×7.25mm(56.95mm^2)
Transistor: 83638

Figure 5.3: The Layout of iitk-RISC.

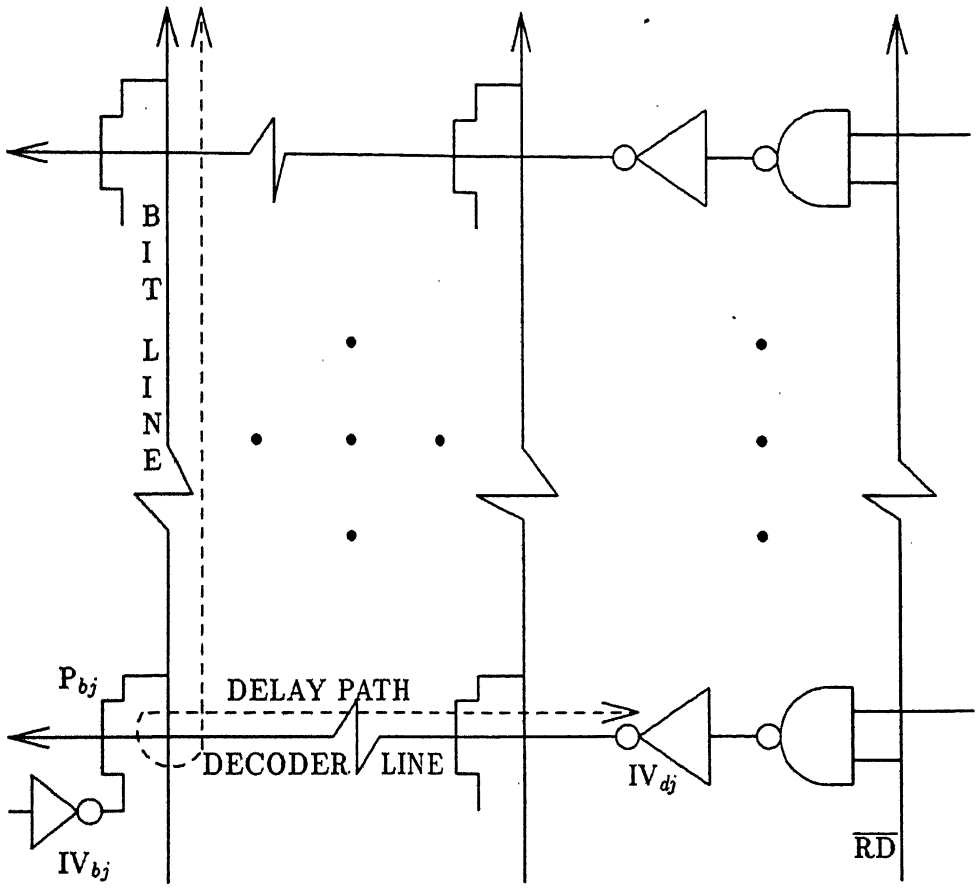
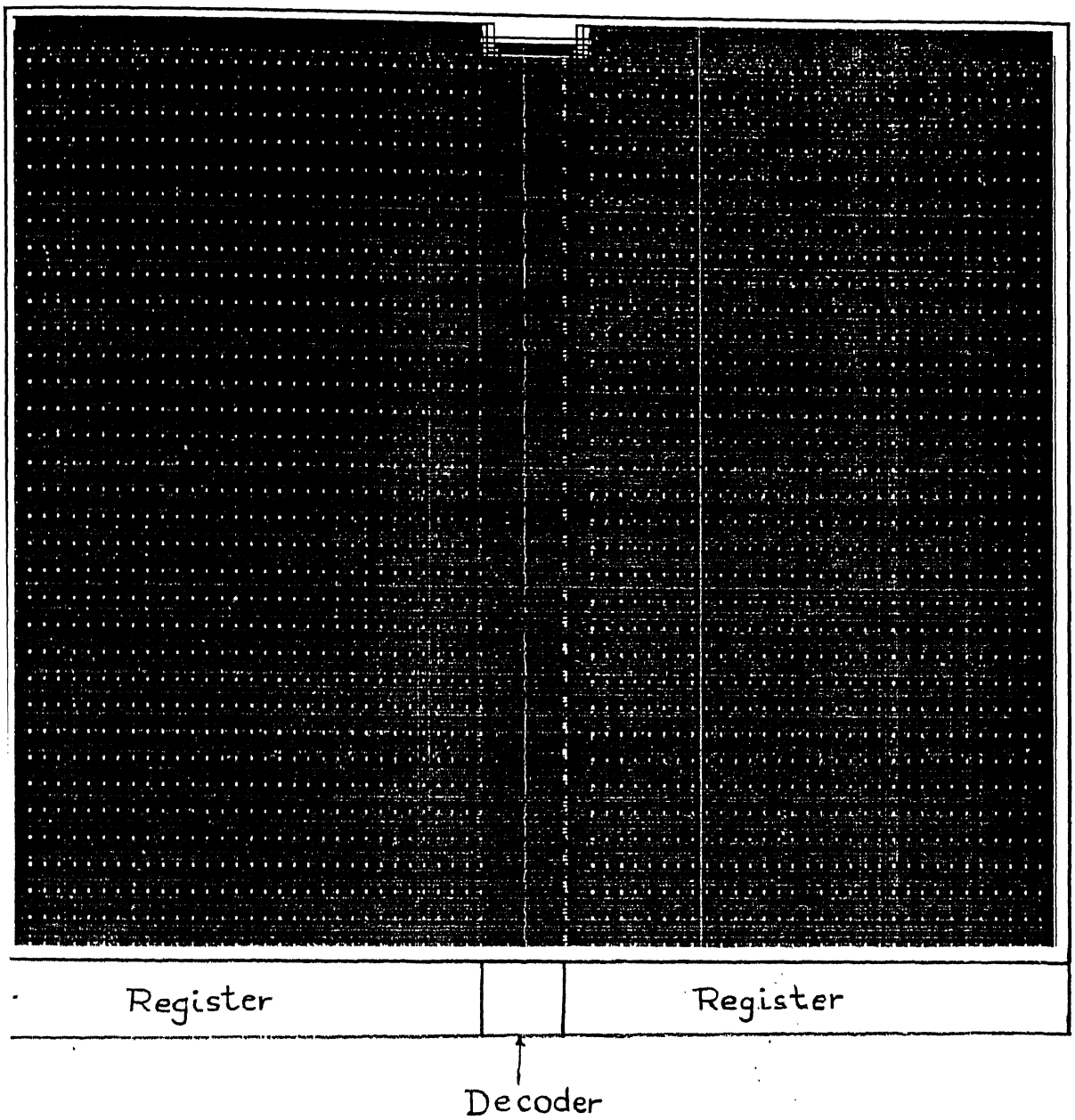


Figure 5.4: The Critical Delay Path of Register File.



Item	Chip area	%Area	Transistors	%Transistors
Registers	20.16mm^2	35.40	57344	68.56
Decoder	1.66mm^2	2.91	3456	4.13
Buffers	0.40mm^2	0.71	1280	1.53
	22.39mm^2	39.32	62080	74.23

Figure 5.5: The Register-file of iitk-RISC.

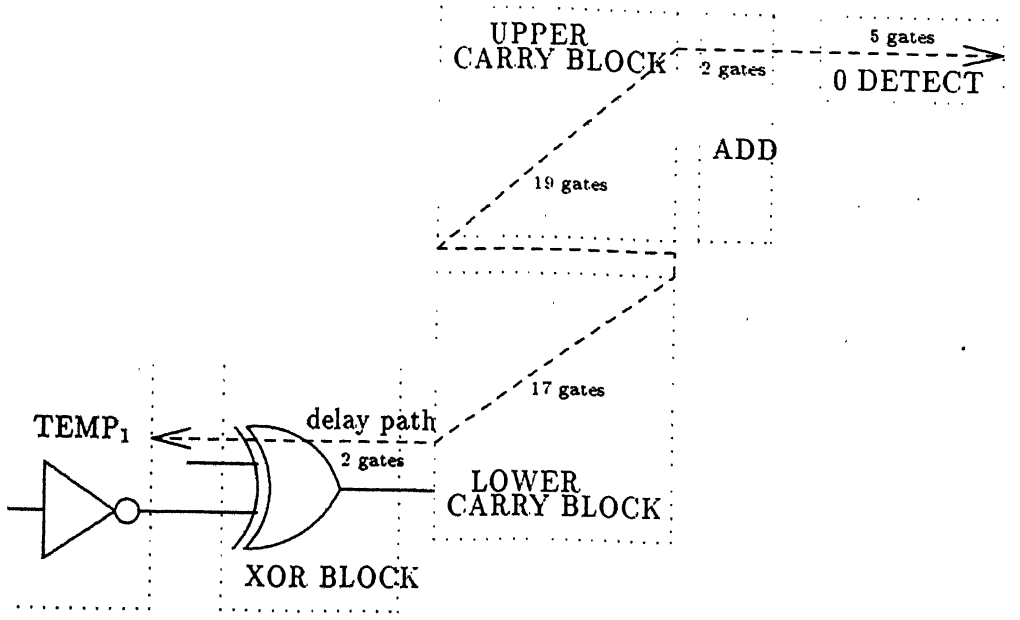
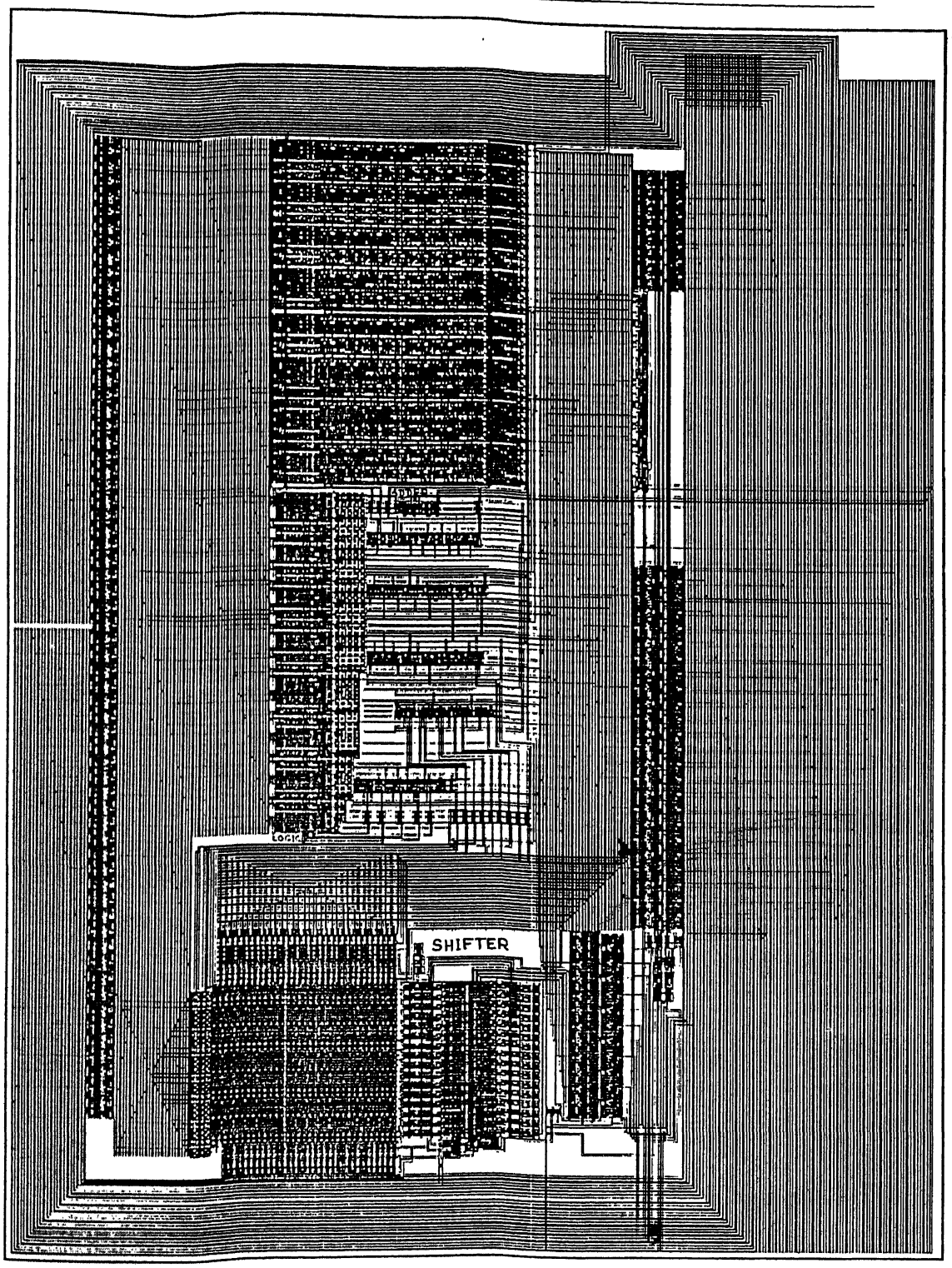
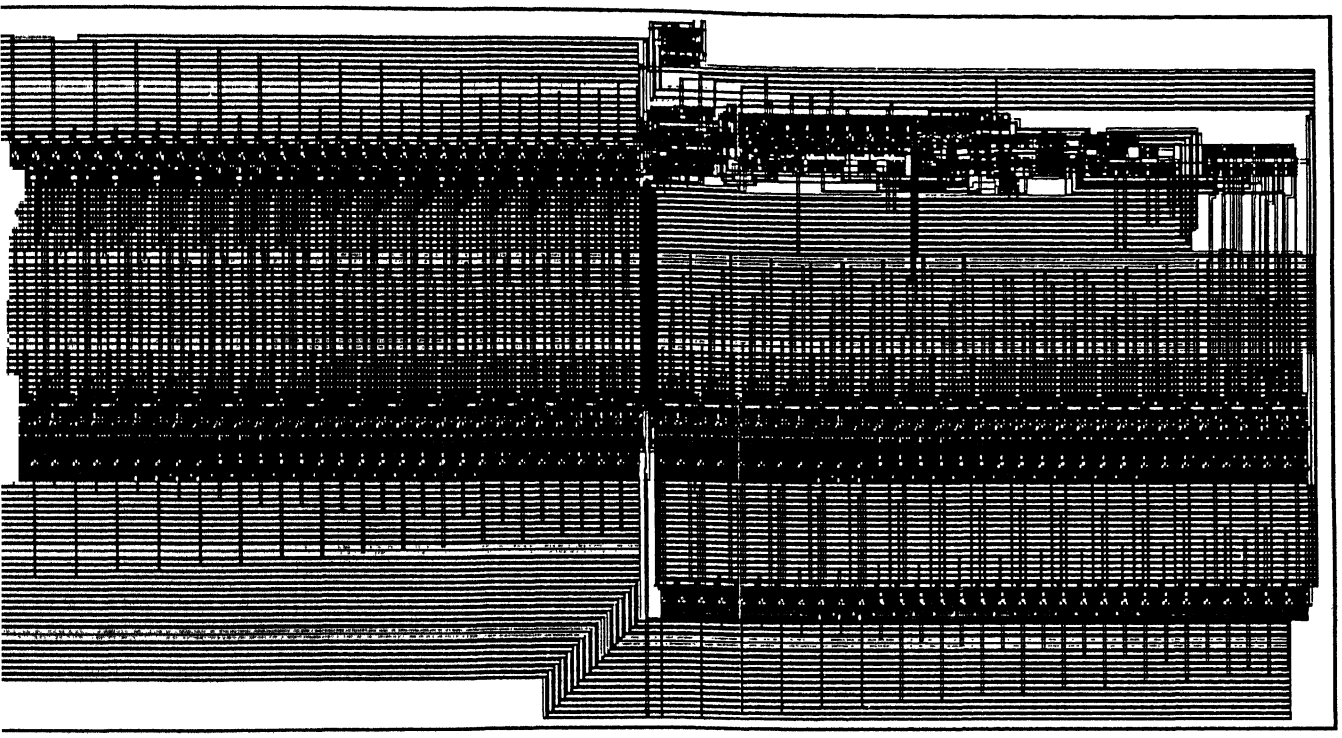


Figure 5.6: The Longest Delay Path.



Item	Chip area	%Area	Transistors	%Transistors
Add unit	1.05mm^2	1.81	2542	3.04
Logic unit	0.54mm^2	0.94	768	0.92
Barrel shifter	1.00mm^2	1.75	1960	2.34
Other control			470	
	7.76mm^2	13.62	5740	6.86

Figure 5.7: The Execution Unit of iitk-RISC.

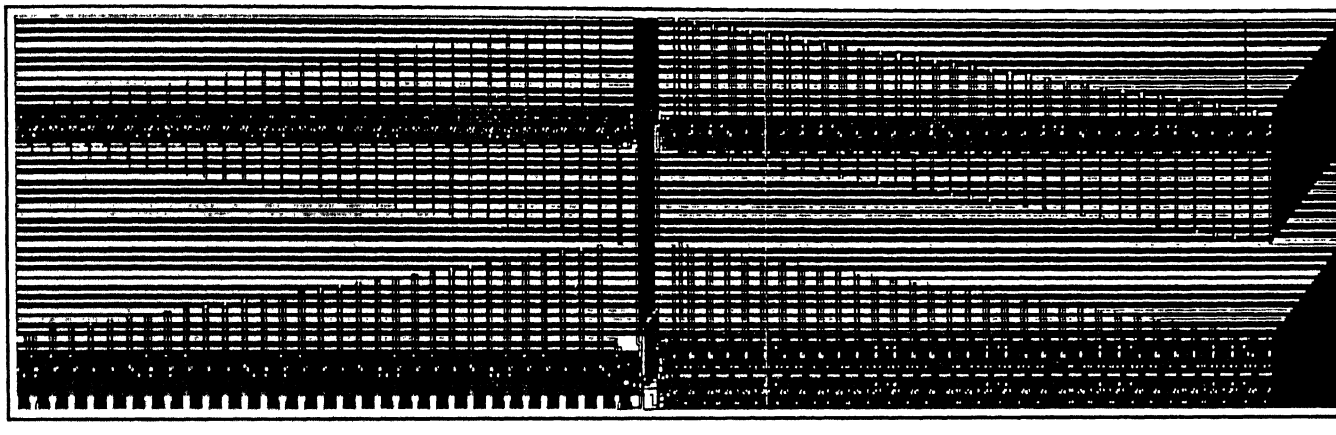


.: MEM_access_unt

0.0 773.71

Item	Chip area	%Area	Transistors	%Transistors
Registers	0.50mm^2	0.88	2624	3.14
Control	0.07mm^2	0.13	338	
Registers	3.89mm^2	6.83	2962	3.54
	4.46mm^2	7.84	5924	7.08

Figure 5.8: The Memory Unit of iitk-RISC.



Chip area	Area	Transistors	Transistors
2.52mm^2	1.43	2096	2.50

Figure 5.9: The Write Back Unit of iitk-RISC.

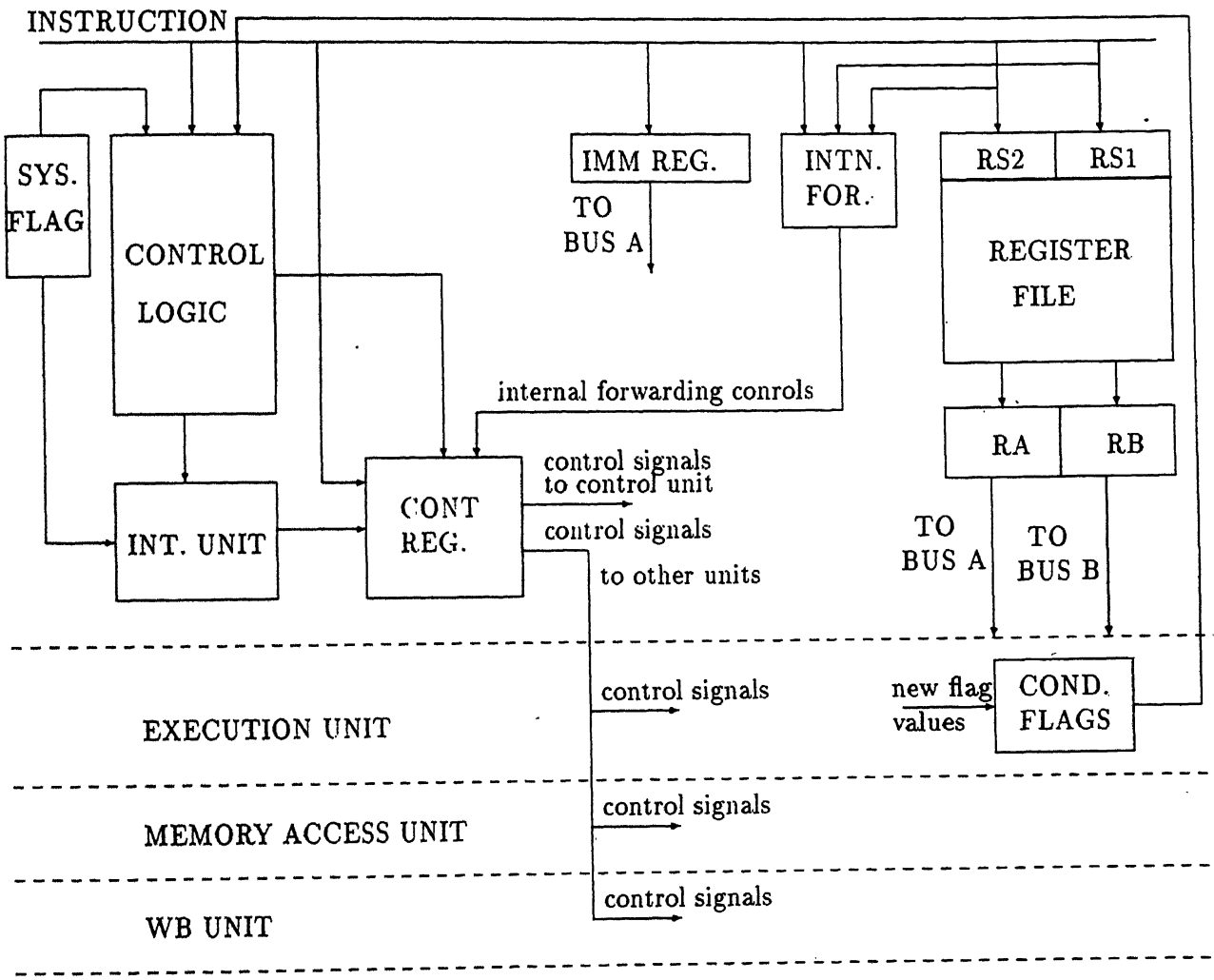
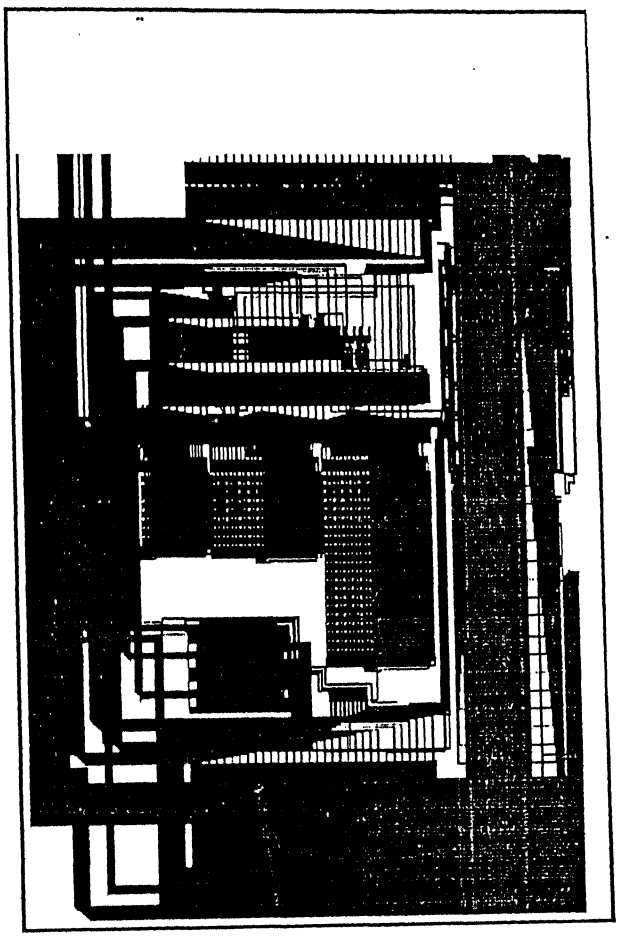


Figure 5.10: Control Path of iitk-RISC.



Cell: control_unit

Item	Chip area	%Area	Transistors	%Transistors
Control logic	1.09mm^2	1.91	2246	2.68
Control Register	0.72mm^2	0.13	1736	2.08
Instruction Registers	0.11mm^2	0.18	448	
Signature Register	0.19mm^2	0.33	896	
LSTPC	0.14mm^2	0.24	512	
PC	1.27mm^2	2.24	874	
	3.52mm^2	5.03	6712	8.03
Others unit:				
Flag register	0.11mm^2	0.20	230	
Immediate register	0.18mm^2	0.31	534	
Z-register	0.01mm^2	0.01	32	
Interrupt unit	0.04mm^2	0.06	90	
Other control			200	
	0.34mm^2	0.58	1086	1.30

Figure 5.11: The Control Unit of iitk-RISC.

Chapter 6

Conclusions

RISC has become a mainstream movement to improve computing power and keep cost of design and design time down. It utilizes the chip area and high bandwidth of on-chip communication effectively. In this thesis a 32 bit VLSI RISC architecture is designed and referred as iitk-RISC.

In chapter 2 some of the existing RISC architectures are surveyed. Following are the interesting features of these RISC processors.

- All are 32 bit architecture.
- All are pipelined architecture and use a 2 to 4 stages pipeline.
- The number of on-chip registers are decreasing. Some new architecture has few on chip registers.
- The size of instruction set and number of addressing modes supported are increased.
- The complexity of the processors is also increased considerably.
- Delay jump concept is used by almost all the RISC architectures.
- On-chip or off-chip instruction and data caches are used by almost all processors.

The above findings led to the formulation of the architectural definition of iitk-RISC. Some of the good features of the existing RISC processors are taken while designing iitk-RISC.

In Chapter 3 the instruction format, addressing mode and instruction set of iitk-RISC are discussed. The iitk-RISC supports an instruction set with fixed fields of fixed size and position. The instruction formats are Register-Register, Long-Immediate and Short-Immediate.

A particular type of instruction format is meaningful only with its counterpart addressing mode. The iitk-RISC instruction set supports Register-Register, Short-Immediate, Long-Immediate, Register-Relative, and PC-relative addressing mode. With a few exceptions all Data manipulation instructions support Register-Register and Short-Immediate addressing, whereas control transfer instructions support Register-Relative and PC-Relative addressing. Instruction set has 57 instructions and it includes instructions for register to register data transfer, data manipulation, memory access and some miscellaneous operations. The memory map of iitk-RISC is 4 Giga byte and organized in bytes banks. Instructions are provided to read a signed or unsigned byte, a signed or unsigned half word, and a full word from memory. Similarly instructions are provided to write a byte, a half word, and a full word into memory. The iitk-RISC CPU has two modes of operation namely privilege and non-privilege. The current mode of CPU is determined by a system flag P.

The pipeline of iitk-RISC, described in chapter 4, is a four stage pipeline. The stages of pipeline include Instruction fetch and decode, execution, Memory access, and WB(write back). Unlike other RISC processors memory access stage is a regular feature of iitk-RISC pipeline. In control transfer instructions control transfer is delayed by one instruction to avoid bubble inside the pipeline.

Instruction is fetched during the IF part of cycle and decoded by control section of control unit. Control section includes control register and control logic. control logic generates all the control signals that may be required during the execution of an instruction. These control signals together with control signals for internal forwarding are pushed in control register. Control register is a three stage register and uses the control signals to the different stages of pipeline during the execution.

The register-file has 128 registers of 32 bit each. Register R_0 is a special type of register and always contains 0. Read and write operations on it are allowed. The execution unit is capable of performing 32 bit operation in single cycle. Conditions flags are modified according to the result of operation performed by execution unit if SCC bit of instruction is set and allowed for the instruction. All memory access are done by memory access unit. If instruction is not a memory access instruction then it introduces a latency and at the end of cycle result is forwarded to write back unit. The write back unit writes the forwarded data into register-file, flag register, or no where depending on the destination of the result, which is determined by the instruction in execution.

Item	Chip area(mm^2)	%Chip area	Transistors	%Transistors
Register file	22.39	39.32	62080	74.22
Execution unit	7.76	13.62	5740	6.86
Memory access unit	4.46	7.84	2962	7.08
Write back unit	2.52	4.43	2096	2.50
Control unit	3.52	5.03	6712	8.03
Others	0.34	0.58	1086	1.30

Table 6.1: Design Metrics for iitk-RISC.

Interrupt unit processes the interrupt requests from off-chip devices. memory access unit on illegal memory address, or control unit on illegal instruction. Off-chip request can be disabled by clearing the condition flag I. Interrupt from memory access unit has the highest priority.

In chapter 5 implementation issues are discussed. The implementation issues includes design of data path, different functional units, and control path. The control unit of iitk-RISC occupies only 6.18% of area. For fast operation of iitk-RISC control logic is designed considering the fact that all the control signals are not required at the same moment. This enables us to break the control logic in parts. Some other optimization technique are also used to further reduce the size of control logic.

In table 6.1 the design metrics of iitk-RISC is given. The layout for iitk-RISC is designed using *c3tu* process, **double metal CMOS technology**, keeping λ at $0.20\mu(1.6\mu$ technology). The total number of pins required are 82.

The iitk-RISC architecture can be made more effective by making some extensions to its present architecture. Some of the suggestions are given during the analysis. We have listed down all of them and some other possible extensions.

- On-chip instruction cache may be used instead of off-chip.
- On-chip or off-chip data cache may be used.
- memory management unit(MMU) may be used.
- In the last, Support for specialized I/O devices like DMA, graphics controller may be provided.

Appendix A

Instruction Set for iitk-RISC

The iitk-RISC instructions are full word long and require one cycle to execute when pipeline is full. The first two columns in table 6.1 give the mnemonic and a brief description of each instruction. The third column gives the opcode of an instruction. The fourth column indicates the addressing modes that may be supported by the instruction using the following symbols:

RR Register- Addressing

LI Long-Immediate Addressing

SI Short-Immediate Addressing

RL Register-Relative Addressing

PL PC-Relative Addressing

The fifth column indicates the possible flag settings using the following symbols:

\- Not affected

0 Cleared to 0

1 set to 1

x set or cleared according to the result of execution unit if SCC=1

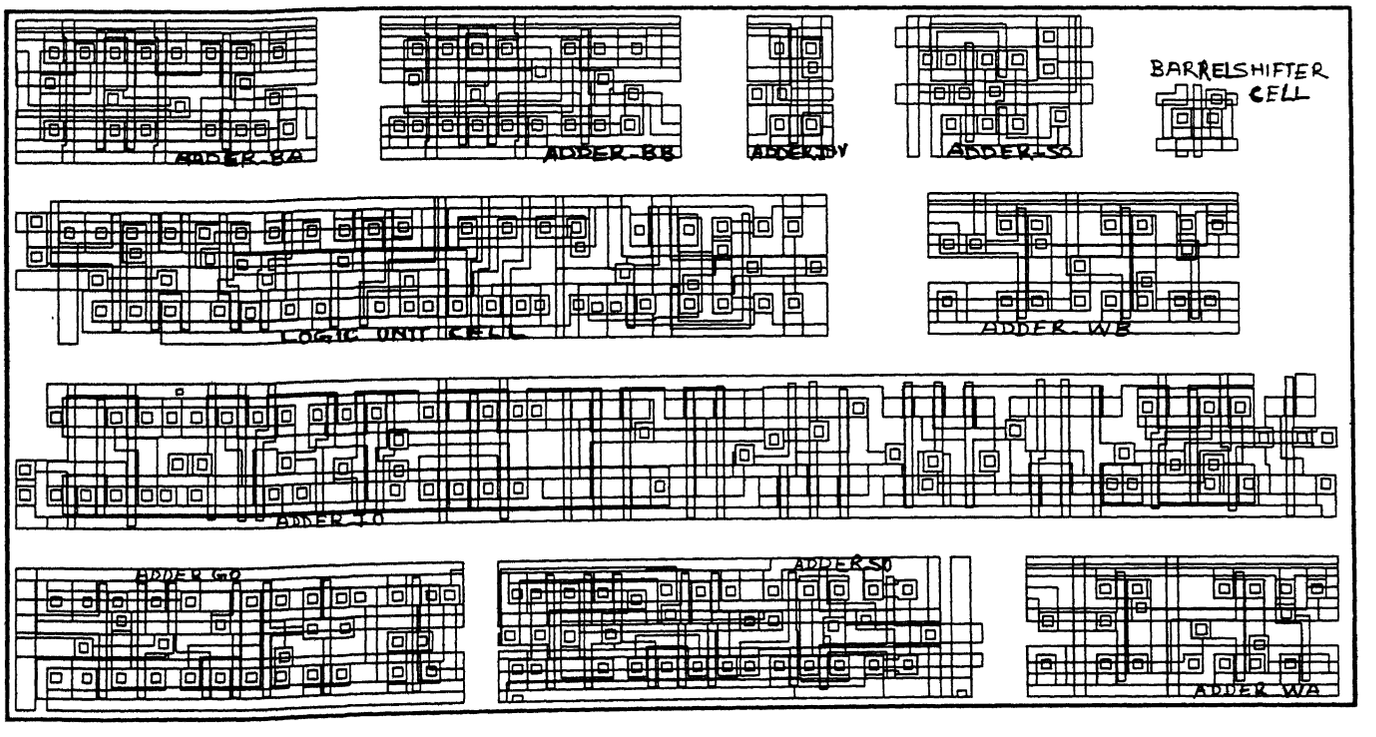
w set or cleared according to data to be written by WB-unit

Mnemonic	Description	opcode	Addressing Modes	flags C	S	Z	V
adc	Add with carry	02H	RR,SI	x	x	x	x
add	Addition	01H	RR,SI	x	x	x	x
and	Logical AND	05H	RR,SI	0	x	x	0
callp	Call Soubroutine	16H	PL	-	-	-	-
callr	Call Soubroutine	15H	RL	-	-	-	-
getlpc	Get last PC	19H	RR	-	-	-	-
getpsw	Get PSW	18H	RR	-	-	-	-
inv	Logical NOT	08H	RR,SI	-	-	-	-
jcp	Jump on Carry	38H	PL	-	-	-	-
jcr	Jump on Carry	28H	RL	-	-	-	-
jep	Jump if Equal	3CH	PL	-	-	-	-
jer	Jump if not Equal	2CH	RL	-	-	-	-
jgep	Jump if Greater or Equal	33H	PL	-	-	-	-
jger	Jump if Greater or Equal	23H	RL	-	-	-	-
jgtp	Jump if Greater	31H	PL	-	-	-	-
jgtr	Jump if Greater	21H	RL	-	-	-	-
jhip	Jump if Higher	35H	PL	-	-	-	-
jhir	Jump if Higher	25H	RL	-	-	-	-
jlep	Jump if Less or Equal	32H	PL	-	-	-	-
jler	Jump if Less or Equal	22H	RL	-	-	-	-
jlosp	Jump if Lower or Same	36H	PL	-	-	-	-
jlosr	Jump if Lower or Same	26H	RL	-	-	-	-
jltp	Jump if Less than	34H	PL	-	-	-	-
jltr	Jump if less than	24H	RL	-	-	-	-
jmpp	Jump always	3FH	PL	-	-	-	-
jmpr	Jump Always	2FH	RL	-	-	-	-
jncp	Jump if No Carry	37H	PL	-	-	-	-
jncr	Jump if No carry	27H	RL	-	-	-	-
jnep	Jump if Not Equal	3BH	PL	-	-	-	-
jner	Jump if Not Equal	2BH	RL	-	-	-	-
jnp	Jump if positive	3AH	PL	-	-	-	-
jnr	Jump if negative	2AH	RL	-	-	-	-
jnvp	Jump if No Overflow	3DH	PL	-	-	-	-
jnvr	Jump if No overflow	2DH	RL	-	-	-	-
jpp	Jump if Positive	39H	PL	-	-	-	-
jpr	Jump if Positive	29H	RL	-	-	-	-
jvp	Jump if Overflow	3EH	PL	-	-	-	-
jvr	Jump if Overflow	2EH	RL	-	-	-	-

Mnemonic	Description	opcode	Addressing Modes	flags C	S	Z	V
loadbs	Load Signed byte	0FH	RR,SI	-	-	-	-
loadbu	Load Unsigned byte	10H	RR,SI	-	-	-	-
loadhs	Load Signed Half Word	0DH	RR,SI	-	-	-	-
loadhu	Load Unsigned Half Word	0EH	RR,SI	-	-	-	-
loadw	Load full Word	0CH	RR,SI	-	-	-	-
mov	Move register	14H		-	-	-	-
nop	No operation	00H		-	-	-	-
or	Logical OR	06H	RR,SI	0	x	x	0
putpsw	Load Flag register	1AH		w	w	w	w
reti	Return From Interrupt	1BH		-	-	-	-
sar	Shift Arithmetic Right	0BH	RR,SI	x	x	x	0
sbb	Subtract with Borrow	04H	RR,SI	x	x	x	x
shl	Shift Logical Left	09H	RR,SI	x	x	x	0
shr	Shift Logical Right	0AH	RR,SI	x	x	x	0
storb	Store a byte	13H	RR	-	-	-	-
storh	Store a Half Word	12H	RR	-	-	-	-
storw	Store a Word	11H	RR	-	-	-	-
sub	Subtraction	03H	RR,SI	x	x	x	x
xor	Logical XOR	07H	RR,SI	0	x	x	0

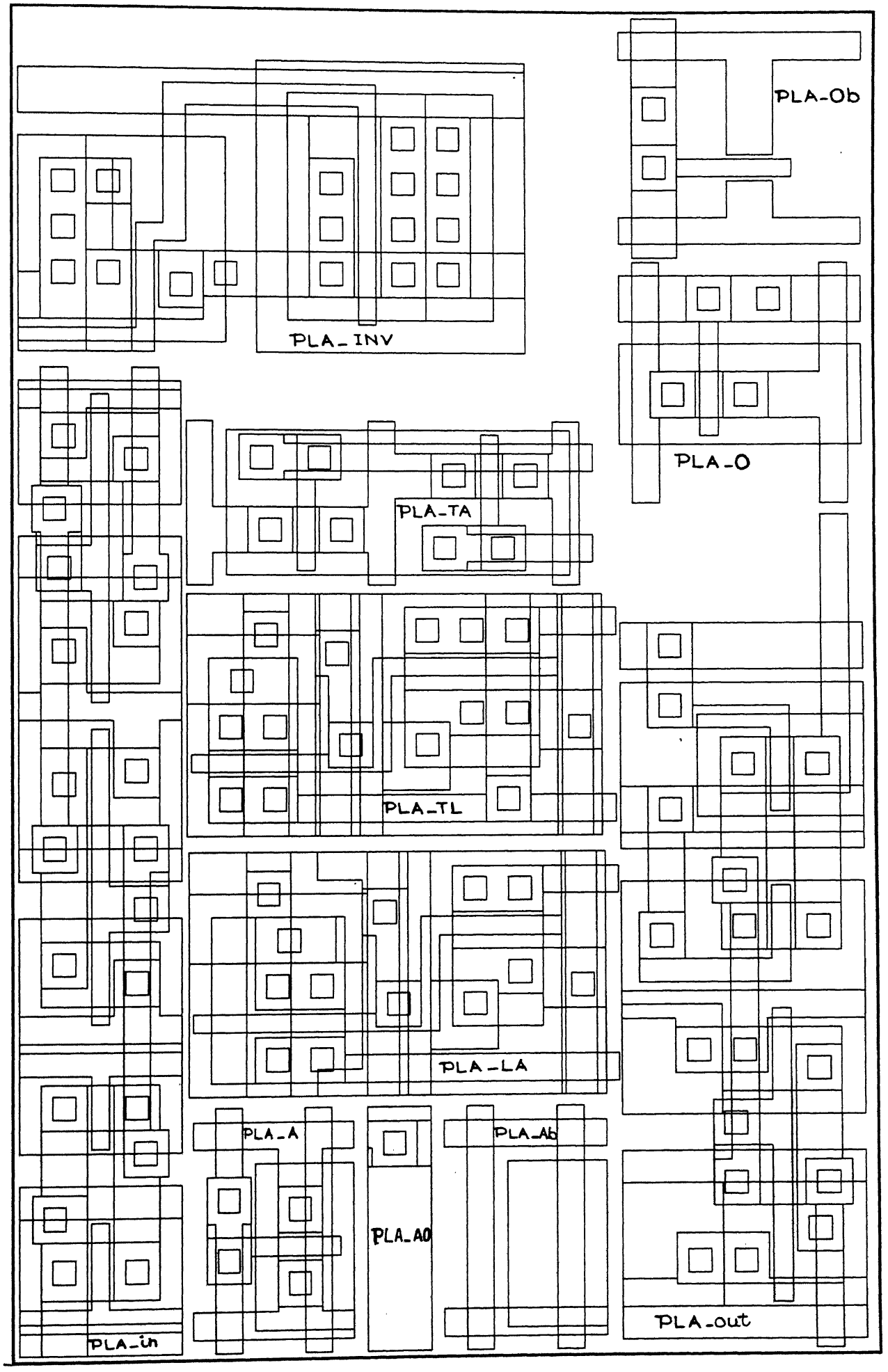
Appendix B

Basic Modules of iitk-RISC Layout

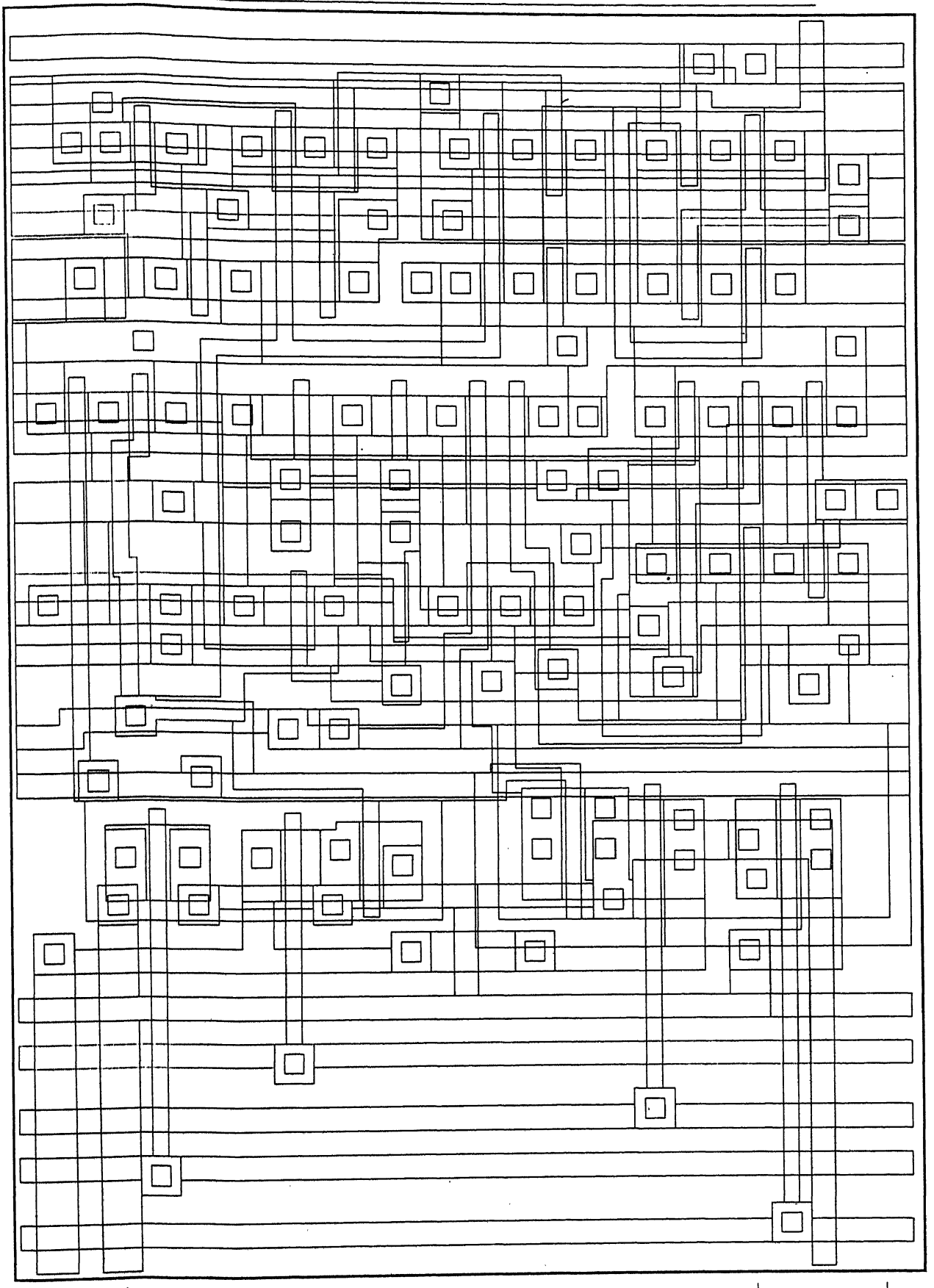


cell: execution_unit_cell

0.0 79.89

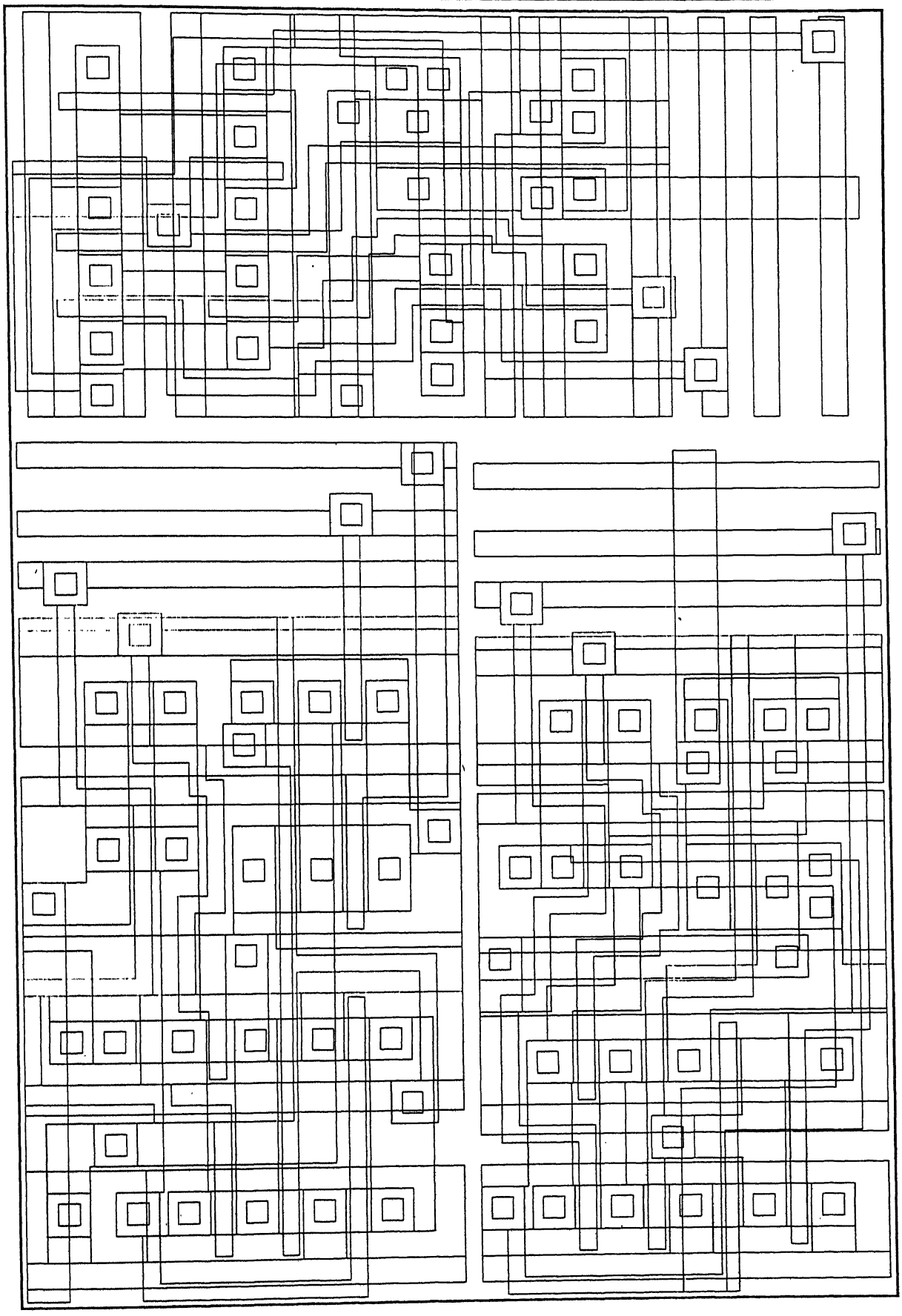


Cell: PLAcells

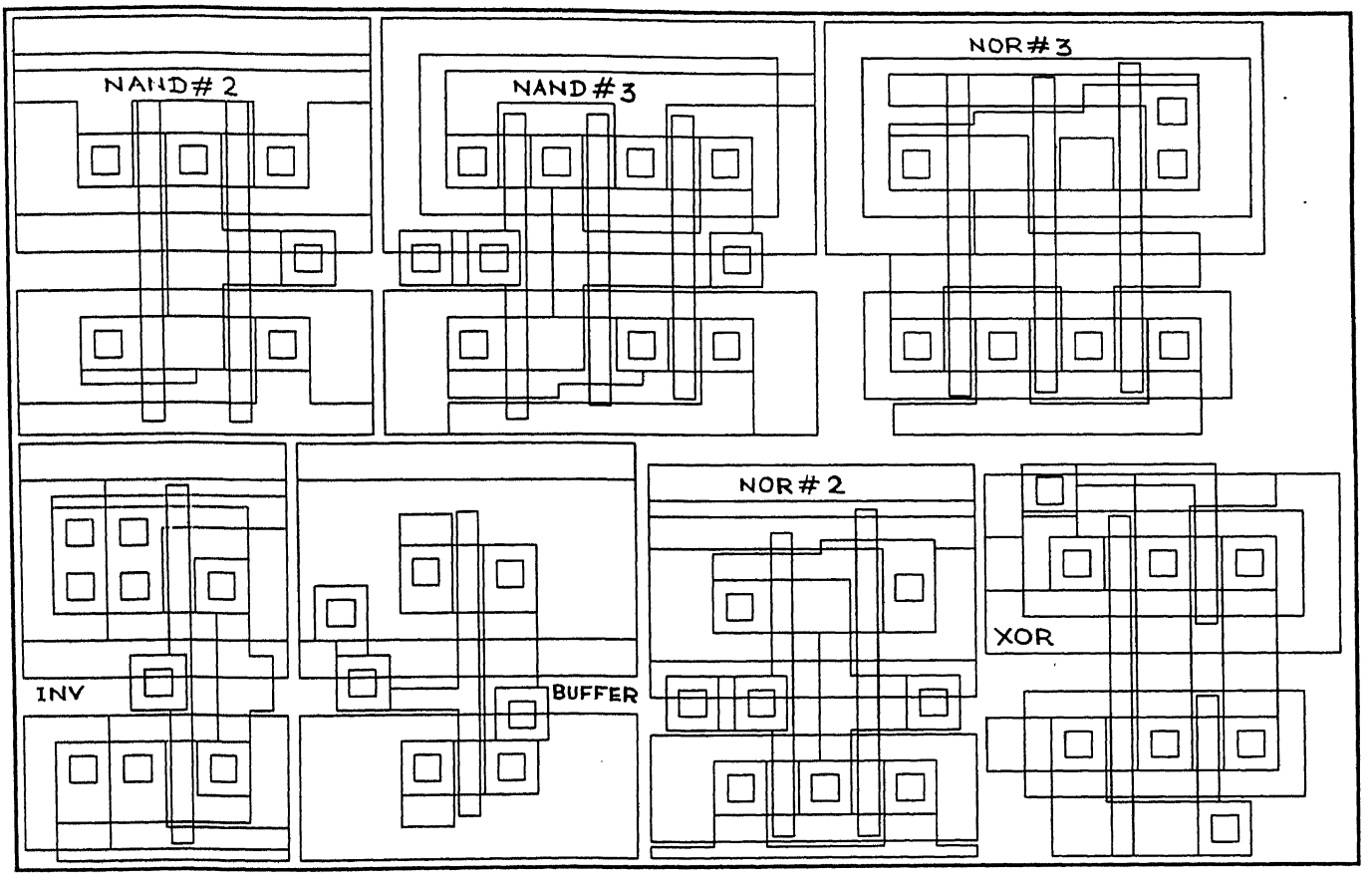


Cell: counter_cell

0.0 25.71

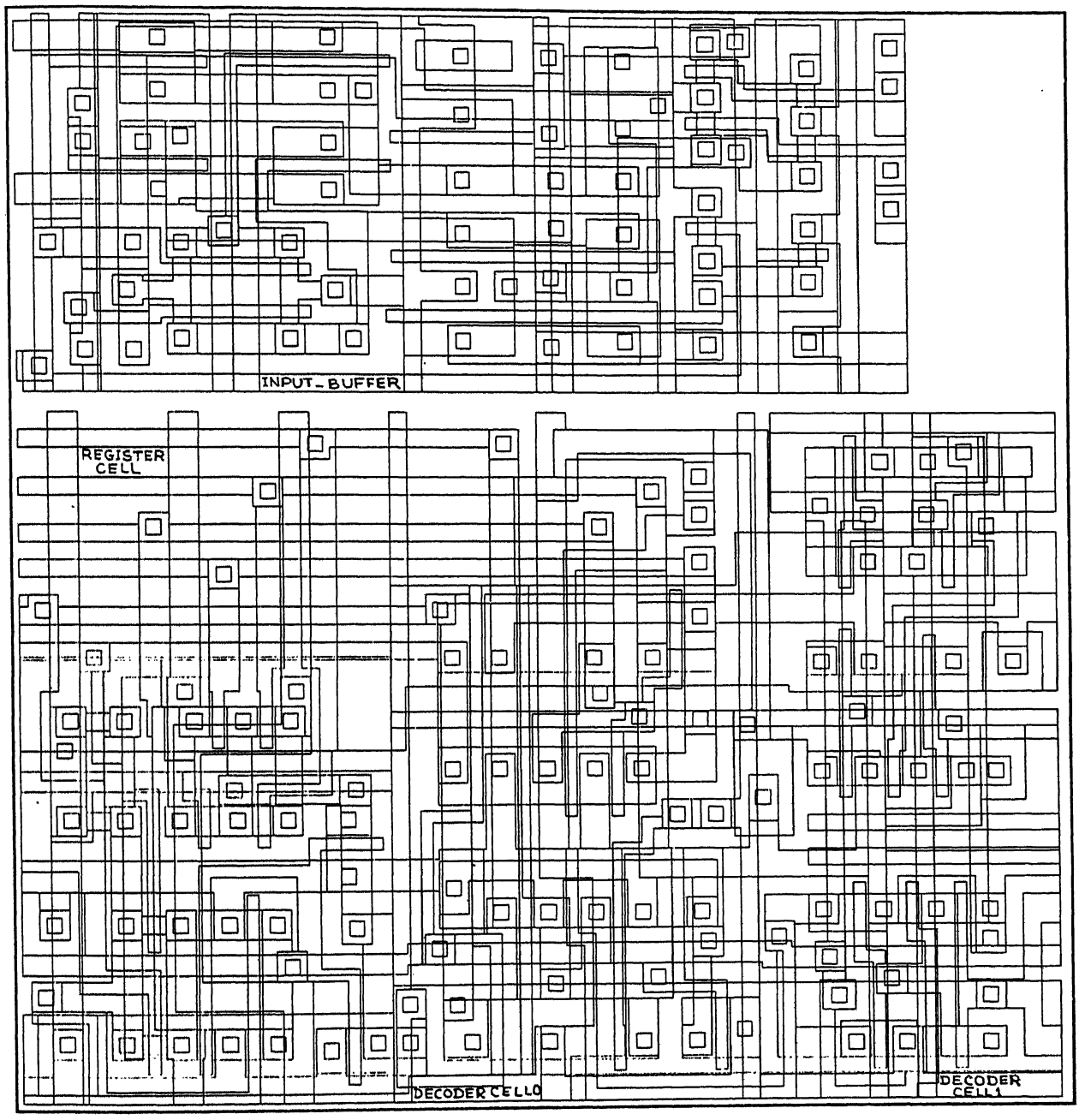


Cell: Dlatches



Cell: gates

0.0 28.29



Cell: register_file_cells

0.0 39.89

Bibliography

[1] Cushman, C. (1988). *"RISC changes the BALANCE"*, VLSI System Design, Vol-6, pp.66-69, June 1988.

[2] Cushman, C. (1988). *"Surveying the RISC Realm"*, VLSI System Design, Vol-7, pp.60-63, July 1988.

[3] espresso on-line manual.

[4] de Graaf, A. C., Van Genderen, A. C. (1988). *"SLS: Switch-Level Simulator"*, The Nelsis IC Design System Documentation, Delft University of Technology, 1988.

[5] Katevenis, M. G. H. (1985). *Reduced Instruction Set Computer Architecture for VLSI.* The MIT Press.

[6] Mead, C. A and Conway, L. (1980). *Introduction to VLSI Systems.* Addison-Wesely, Reading, Mass.

[7] "The Nelsis IC Design System Users' Manual," TU Delft Software Distribution, Delft University of Technology, April 1989.

[8] Patterson, D. A. and Sequin, C. H. (1982). *"A VLSI RISC"*, IEEE COMPUTER, Vol.9, pp.8-20,September 1982.

[9] Rowen, C. Freitas, D. Hansen, C. Hudson, E. Kinsel, J. Moussouris, J. Prizybylski, S. and Riordan, T (1986). *"RISC VLSI Design for System-Level Performane"*, VLSI System Design, Vol-3, pp.81-88, March 1986.

[10] SPICE3d2 User manual.

[11] Weste, N. H. E. and Eshraghian, K. (1985). *Principles of CMOS VLSI design: A system perspective Apporach.* Addison-Wesely Publishing Company.